UCRL-53190

# On the Design of High Performance Digital Arithmetic Units

Paul Michael Farmwald

(Ph.D. Thesis)

August 1981

Lawrence Livermore National Laboratory

UCRL-53190
Distribution Category UC-32

# On the Design of High Performance Digital Arithmetic Units

Paul Michael Farmwald

(PH.D. Thesis)

Manuscript date: August 1981

LAWRENCE LIVERMORE LABORATORY
University of California • Livermore, California • 94550

# On the Design of High Performance Digital Arithmetic Units

A DISSERTATION

SUBMITTED TO THE COMPUTER SCIENCE DEPARTMENT

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

by

Paul Michael Farmwald

August 1981

## Acknowledgments

I would like to thank Forest Baskett, Joseph Oliger and Lowell Wood for the support and guidance they have provided during the course of this work. In particular, without the constant "gentle" prodding of Lowell, this work would have taken much longer. The support of the Fannie and John Hertz Foundation during the course of this work was very valuble in giving me the freedom to pursue this research. I wish to thank Bill Bryson who was the co-designer of S-1 Mark IIA ABOX, and Jeff Rubin and Tom McWilliams, who designed the IBOX. Many others provided valuable help and discussions, including Allan Miller, Charles Frankston, Dan Weinreb, Dave Goldberg, Dick Gabriel, Don Knuth, Earl Killian, Erik Gilbert, Hans Moravec, Hon Wah Chin, Jeff Broughton, John Manferdelli, John McCarthy, Lansing Sloan, Larry Robinson, Rod Hyde, Rodney Brooks, Steve Correll, Ted Anderson, and W. Kahan. I also wish to thank Chris Ghinazzi, Paula Berman, and Gloria Hernandez for helping with all the last-minute preparations.

This work was in part performed under the auspices of the U. S. Dept. of Energy at Lawrence Livermore National Laboratory, which is operated by the University of California under contract No. W-7405-ENG-48.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Forest Baskett (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Joseph Oliger

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Lowell L. Wood

Lowell L. Wood

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

# Contents

# 1. Introduction

Ever since the electronic digital computer age commenced several decades ago, digital computer designers have sought to make hardware which can execute programs at ever faster rates. Two basic approaches to the goal of higher computing throughput have been pursued: use of more modern components which perform digital logic functions more rapidly and more cheaply than the ones which they replace, and the use of components of a fixed technological level in a more effective fashion. This thesis is almost exclusively concerned with the latter; the former is of concern only where recent technological advances have impacted the relative desirability of certain choices, or where the algorithms used affect packaging decisions.

The author has designed the arithmetic hardware of the S-1 Mark IIA uniprocessor system. The Mark IIA average throughput approaches that of a Cray-1 when doing floating-point-intensive computations, but also computes at very high performance levels when doing most other types of modern digital data processing. The goal of the Mark IIA design effort has been to realize a computing system with its capabilities distributed in a balanced fashion throughout the spectrum of modern digital computing interests. Several of the techniques used in this effort at balanced high-performance processor design are original and are described herein. In addition, an attempt has been made to describe why these techniques are useful and, since their relative values often depend upon choices that were made during the definition of the architecture to be implemented and of its high-level design, the rationale underlying these decisions is also sketched.

## 1.1 Outline of the Thesis.

Chapter 2 discusses pipelines and pipelined general-purpose computers. It defines terms and provides a basis for the discussion in the remainder of the thesis.

Chapter 3 is a somewhat detailed description of the S-1 Mark IIA. It characterizes the framework within which the contributions of this thesis have been made, as well as motivating many of the examples in later chapters.

Chapter 4 resolves computer systems into two classes: those in which the hardware uses the same representation for storage and for computation and those which allow the two to be different. The advantages of the latter type of system are described, with examples taken from the S-1 Mark IIA.

Chapter 5 describes a new floating-point addition algorithm which has a significantly shorter latency than any previous ones which the author has been able to discover. In addition, the technique used in the algorithm permits the simultaneous computation of the floating-point sum and difference of two operands for much less than twice the hardware cost of a single floating-point adder; such computations are highly useful in performing many important algorithms such as the FFT.

Chapter 6 describes an algorithm for very fast computation of elementary functions to medium precision. The method is applied to computation of the functions for reciprocal, square-root, exponential, logarithm, arctangent, sine, cosine, and error function. In essence, the method permits the computation of a function with $p$ bits of accuracy using two $p$-bit multipliers and approximately $3p2^{\frac{p}{3}}$ bits of table look-up memory, and requires very little more than a single multiplication time to execute. The S-1 Mark IIA uses 1K RAM chips to achieve an accuracy of 30 bits of precision for each of these functions, and also provides iterative means for increasing precision to twice this level. The size of high-performance RAMs are improving at rates comparable to most of the rest of semiconductor technology; the maximum size RAM available doubles every two to three years, implying that use of this algorithm will permit computation with an extra bit of precision every year! Therefore, this technique may well become the method of choice for elementary function evaluation on future computer systems.

Chapter 7 presents an extension of the technique of skewed storage representations in interleaved memory systems. Rather than skewing the representation in main memory, a small memory is used to skew the data "on the fly" during processing. Thus, standard representations of data may be employed while nonetheless obtaining the efficient processing benefits of skewed storage. This extended technique has an important collateral advantage, as the hardware used to implement it can double as a throughput-enhancing buffer between memory and the arithmetic processing element.

Chapter 8 derives several algorithms (which are based upon Quicksort) for sorting of arrays on a parallel pipelined system. This algorithms allow the efficient use of interleaved memory systems and pipelined CPUs.

Chapter 9 is the "implementation" section. It first describes why high-performance arithmetic functional units (as well as other pipelines which have certain general characteristics) should be thought of as computations on cylinders, and then presents a timing method for the pipeline latches which greatly simplifies the distribution of certain global control signals.

While Chapters 3 through 9 describe techniques actually used in the design of the S-1 Mark IIA uniprocessor, Chapter 10 describes a technique that the author would have used in this system, had the design schedule permitted. It is an algorithm for out-of-order execution of instructions that doesn't suffer from imprecise interrupts. It involves a small memory logically located just before the final output of the arithmetic hardware that is used to re-order the results of computations. Thus, from its outside, the arithmetic section appears to be a strictly in-sequence execution unit (and can correctly recover from arithmetic faults); internally, however, it can and does obtain all of the the overall performance advantages of an out-of-order pipeline control mechanism.

Chapter 11 summarizes the contributions made by the research whose results are documented in this dissertation, and discusses directions considered to be fruitful for further work in this area.

# 2. Preliminaries

Much of the work discussed in this thesis involves the concept of pipelining (of both SIMD and SISD systems [Flynn72]). The reader is assumed to be familiar with general concepts of pipelined systems and especially pipelining in SISD systems. ([Kogge81a] is an excellent introduction to the subject.) We define a few terms that might otherwise be unclear. The *latency* of an operation (for our purposes) is defined to be the time from when the operation is "started" to when an immediately following operation dependent upon the first is "started". With this definition, an operation may have many latencies, one for each possible way other operations could be dependent upon it. To make this more concrete, consider the latencies possible with an integer multiplication instruction. On the Mark IIA, which does shortstopping within the execution unit, the latency of such a multiplication is three cycles with respect to a succeeding integer operation, ten cycles with respect to a succeeding operation that uses the result of the multiplication as an index, and thirteen cycles with respect to a operation which conditionally branches on the result of the multiplication. (Of course other operations can occur "in between" such dependent operations).

A simplified diagram of a system which exhibits these latencies is shown in Figure 2.1, and serves to show the conventions which we will use to draw pipelines in this thesis. With these conventions a pipeline may have fixed delay ("LENGTH=2") or variable delay ("2≤LENGTH≤ 3"). Instructions leave a pipeline in the same order as they entered, unless explicitly noted otherwise. Whenever the output of a pipeline joins the input to a pipeline (including itself), a dependency may result (which defines a latency!).

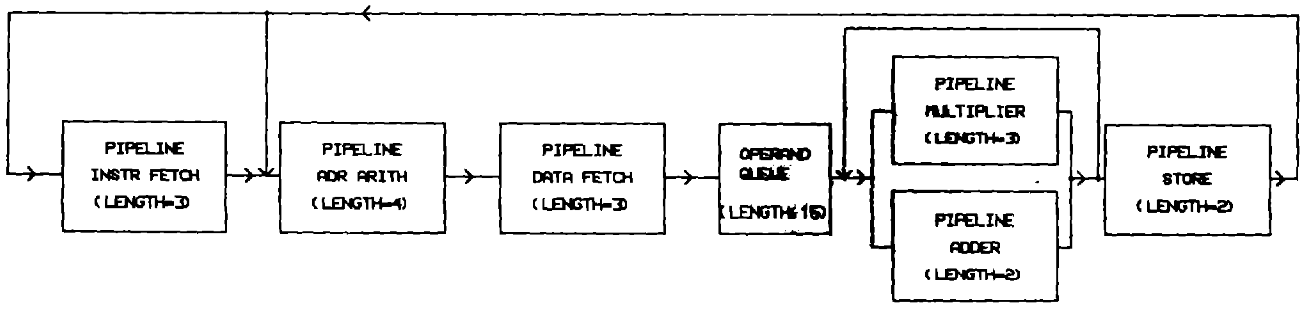


**Figure 2.1:** Simple Pipelined System

# 3. The S-1 Mark IIA Uniprocessor

In order to additionally motivate the results of this dissertation, a brief description of an actual computer on which they have been implemented, the S-1 Mark IIA, is appropriate.

The S-1 Mark IIA uniprocessor is the second implementation of the S-1 architecture [S-1 Project 79]. The first implementation, the S-1 Mark I, has been operational since mid-1978. It is a moderately high performance scalar processor, with roughly the same throughput as an IBM 370/168. (The Mark I has a 100 nanosecond cycle time and many of its instructions can execute in a single cycle.) It is constructed on twelve large wire-wrap boards containing 5300 ECL-10K chips. Originally, it was planned to build a multiprocessor containing sixteen Mark I's, but it became clear (at least to the author and his colleagues in the S-1 Project) that while such a system might be interesting as a research tool, it would not be the most cost-effective way to achieve a significant advance in the general-purpose compute power available for single tasks. Instead, it was decided to build a much higher performance uniprocessor (the Mark IIA), and to construct a multiprocessor with sixteen Mark IIAs. The first Mark IIA is scheduled to be operational in early 1982. It consists of roughly 25000 ECL 10K and 100K chips interconnected on 64 wire-wrap boards. The main improvements implemented in the Mark IIA (relative to the Mark I) are reduced cycle time, vastly improved arithmetic operation bandwidth, vector operations, larger caches and (of course) much cleverer arithmetic algorithms.

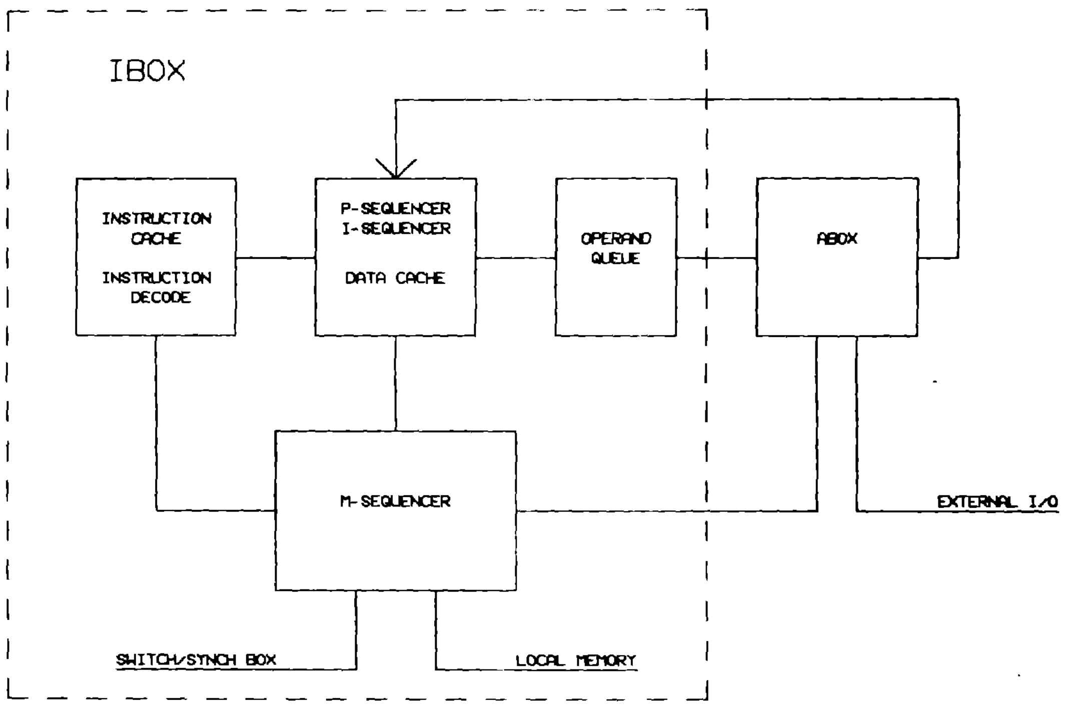


**Figure 3.1:** The S-1 Mark IIA Pipeline

The Mark IIA is divided into two main sections: the IBOX and the ABOX (see Figure 3.1). The IBOX fetches instructions and data and detects hazards in the instruction stream, while the ABOX performs the arithmetic and logical operations on the data (taking into account hazards detected by the IBOX). Both the IBOX and ABOX operations are extensively microcoded, with a total of eleven sequencers, each controlling one or more stages of pipeline, with a separate sequencer to handle cache misses and memory accesses. Sequencers that control more than one pipeline stage are nearly always ***data stationary*** ([Kogge77]), since control for later pipeline stages is delayed along with the corresponding data. However, instead of directly delaying control bits, many Mark IIA sequencers instead delay addresses to microcode-containing RAMs in later pipeline stages.

## 3.1 The S-1 Mark IIA IBOX.

The IBOX pipeline is shown in Figure 3.1. Note that there are separate instruction and data caches. This has the advantage of increasing overall cache bandwidth, as well as the somewhat less obvious benefit of allowing specialization of the two memories. In particular, the instruction cache does "preprocessing" on the instructions at cache miss time (which slightly increases the total latency of a cache miss, but which greatly decreases instruction decoding time), while the data cache is optimized for bandwidth and minimum cache miss execution time.

Although the Mark IIA is capable of executing an instruction every cycle, extensive conditional branching with its relatively long pipeline could limit its performance on "real" programs. In general, branching on pipelined SISD machines presents two problems. The first (common to both conditional and unconditional branches) is concerned with fetching the target instructions without delaying the pipeline. Since the Mark IIA has a large instruction cache (containing pre-decoded instructions), it can nearly always determine the target instruction and fetch it in a single cycle;

thus this aspect of branching is not a problem. However, the second problem is that for conditional branches it is not clear which target is correct! The Mark IIA predicts a branch dynamically (henceforth called ***branch prediction***); if the prediction is incorrect (which is determined at the very end of the pipeline by the ABOX), all subsequent instructions-in-process are flushed from earlier stages of the pipeline and the alternate path is followed. The prediction is based upon the instruction opcode, the most recent branching behavior of that particular instruction location (a single bit is kept for each instruction in the instruction cache, so that the recent branching history of an instruction is forgotten if it is ejected from the cache), and the direction of the branch.

As described in chapter 2, the latency of arithmetic operations with respect to address computations is relatively large. To reduce the performance degradation that might otherwise occur (since this form of dependency is relatively common), the IBOX predicts the values of many simple computations (hence the term ***value prediction***). For instance, consider an integer addition instruction whose operands are in registers (or are constants encoded in the instruction itself) and whose result is to be stored in a register. On the Mark IIA, the actual addition is done twice: first early in the address arithmetic segment of the pipeline, and then later in the ABOX. The result of first addition is remembered as the current (but temporary) value of the destination register (which allows succeeding instructions to use the result of the addition in address arithmetic without the time-delay penalty which would otherwise be imposed) while the ABOX result permanently changes the register. The temporary nature of such value predictions allow the straightforward recovery from incorrectly predicted conditional branches.

The IBOX detects hazards in the instruction stream by using an array of address comparators. All writes must be "scheduled" by placing the address of the write destination into the comparator array. All succeeding instructions can detect hazards by comparing read addresses (of register or cache reads, both for address arithmetic and operands) to all pending writes. (Since the pipeline never reorders the instruction stream, the write-after-write (WAW) hazard is impossible.) When a hazard is detected, the stage exhibiting the hazard is held until the hazard disappears (i.e., the result appears!), except for two special (but nonetheless quite common) cases. First, if the hazard is due to an index register read which has been value-predicted, then the predicted value is used. Second, if the hazard is due to an operand read (i.e., data to be sent to the ABOX to be computed on are unavailable because the ABOX is now computing them) and the read and write can be conformed (i.e., the ABOX tag types match, as discussed below and in Chapter 4), then the (incorrect) data are sent to the ABOX but with extra information sufficient to allow the ABOX to find and use the correct data (which it is about to compute). Note that no mention has been made of the hazard of writing into the instruction stream. This hazard is obviated by a hardware-enforced prohibition against an location being present in the instruction cache and simultaneously being writeable in the data cache.

The Operand Queue between the ABOX and IBOX serves to decouple the effects of transient processing slowdowns in either of the two units. The data cache on the the Mark IIA is capable of reading (and writing) up to sixteen sequential nine-bit bytes every cycle. This high bandwidth is used (among other things) to implement vector operations. The Operand Queue reorders the data from the cache into streams suitable for computation. The sequential nature of the Mark IIA data cache could have been a problem for many algorithms. Chapter 7 describes a method to solve some of these problems.

## 3.2 The S-1 Mark IIA ABOX.

Of the various latencies of an operation, by far the most important (because it comes into play the most often) is the time from when the data are available to begin an operation to the time when the operation's result is available for use by a later operation. The ABOX pipeline (shown in Figure 3.2) reflects an attempt to minimize this latency. *All* operations must pass through at least one of the two functional units (the Adder and the Multiplier). To reduce latency as low as possible, a small crossbar network allows the outputs of both functional units to be "wrapped" back into the inputs of these functional units, thus bypassing the otherwise necessary store to and fetch from memory operations. Some form of wrapping (also known as "shortstopping") is employed on nearly all very high performance machines, sometimes using a high-speed register file instead of a crossbar.

As will be discussed in greater length in chapter 4, the ABOX uses different formats for processing data internally and for storage in memory

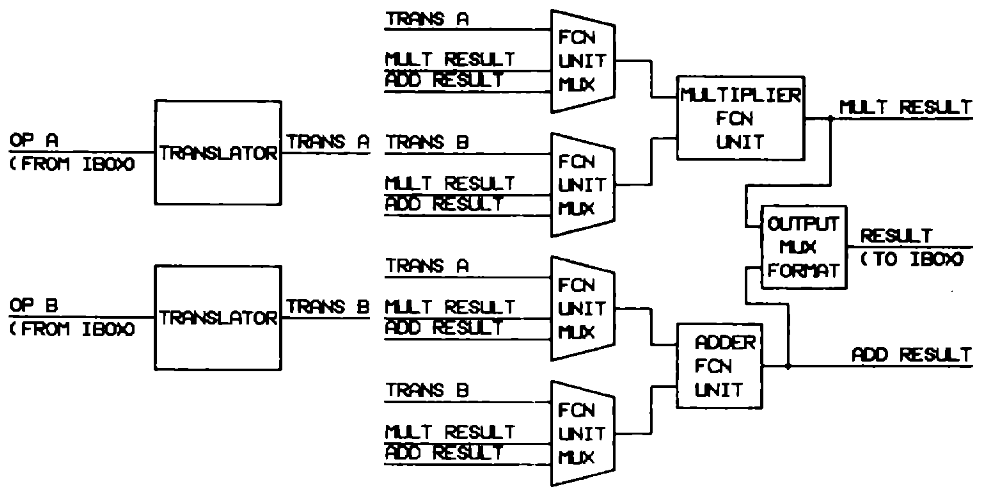


**Figure 3.2:** S-1 Mark IIA ABOX Pipeline

or registers. All internal (to the ABOX) formats are mapped into an internal bus, and for wrapping to occur the same internal format must be in use. The test for compatible formats is included in the IBOX hazard detection circuitry, as mentioned above. (If the formats are not compatible, the IBOX must wait for the ABOX to write the data back into memory.)

When the IBOX detects that wrapping is necessary, it records along with the (invalid) operand the fact that the operand is invalid (by setting the "Wrap" bit) and a number indicating which operation's result yields the correct data (the "Wrap Num"). Thus, if we are attempting to use the result of the previous instruction, Wrap Num will be zero, while for the second previous instruction the number will be one, etc. Internally, the ABOX keeps the last sixteen results it has generated in a RAM (the Wrap RAM) and also maintains three numbers which it uses to determine where to find an operand. The three numbers are (1) the number of results not in the Wrap RAM, (2) the number of results not on the output of the Multiplier Functional Unit or in the Wrap RAM, and (3) the number of results not on the output of any functional unit or in the Wrap RAM. By doing three (parallel) arithmetic comparisons, it can determine the location of the correct data or that they have not been computed (in which case the ABOX must wait for previous operations to complete).

All results (meaning changes in the state of the machine) must pass through the ABOX. Furthermore, in the Mark IIA all such state changes occur in strictly the same order as in the instruction sequence. This admits ***precise interrupts*** in a fairly straightforward way, but can have adverse impact on performance; this is discussed in much greater detail in Chapter 10.

There is one aspect of the Mark IIA control structure about which the author is inordinately proud (even though he realizes that it is probably a fairly minor point in a practical sense). Since the Add Functional Unit delay is two cycles while the Multiplier delay is three cycles, a conflict can occur (in scalar mode) if a multiply operation is followed immediately by an add class operation, since both the add and multiply results would need to be output simultaneously. This was unfortunately out of the question, and so the initial design detected this case and inserted a null cycle so that the add operation finished after the multiply. However, it was observed that the multiplier was idle during the add operation and thus, if the output of the adder could be passed into the *middle* of the multiplier pipeline (which it could), an add operation preceded by a multiplication could be treated as a second multiplication. This avoids the null cycle, and also permits the result of the addition to be available as early as possible for wrapping (since it first appears on the output of the adder, where it is first available for wrapping, and then appears on the output of the multiplier on the next cycle, where it is actually output, and is again available for wrapping).

# 4. Data Formats

Data formats most convenient for use by a programmer (or compiler), and especially those which pack the maximum information in a computer word, are not always the most convenient for use by the arithmetic hardware. The arithmetic hardware often greatly benefits (both in speed and size) by introducing uniformity and redundancy in the representation that it uses for computation.

## 4.1 The Distinction between Internal and External Formats.

In the following discussion, we will refer to the format(s) that the architecture defines (and the user sees) as the *external format* and the format(s) that the hardware actually uses as the *internal format*. Figure 4.1 shows a diagram of a CPU where no such distinction is made, while Figure 4.2 is a CPU where the two formats can be different.

First of all, the internal format must be a superset of the external format, in the sense that it must allow the *efficient* implementation of the operations as defined by the architecture (which by definition are operations on the external formats).

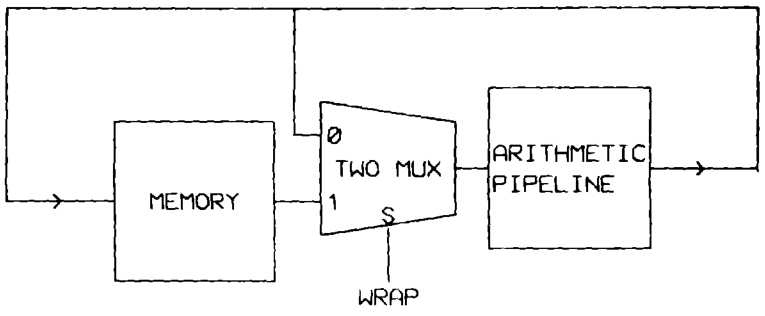


**Figure 4.1:** Block Diagram Of A System Which Uses The Same Formats For Storage And Computing

Furthermore, it should be possible to convert quickly and easily between the two classes of formats, since this will be done so often. However, the whole point of making the formats different is to allow the arithmetic operations to take place in the shortest possible time. This seeming inconsistency disappears if one considers that a short latency time for arithmetic is important on a pipelined machine precisely when the result of an operation is used by a "nearby" instruction in the instruction stream. Since operations on the external format are implemented using the internal format, we can (by directly connecting different stages of the pipeline) "wrap" the needed result from the first place where it is available ***in the internal format***. Thus conversion is done only when it doesn't affect the latency of (nearby) operations in the instruction stream.

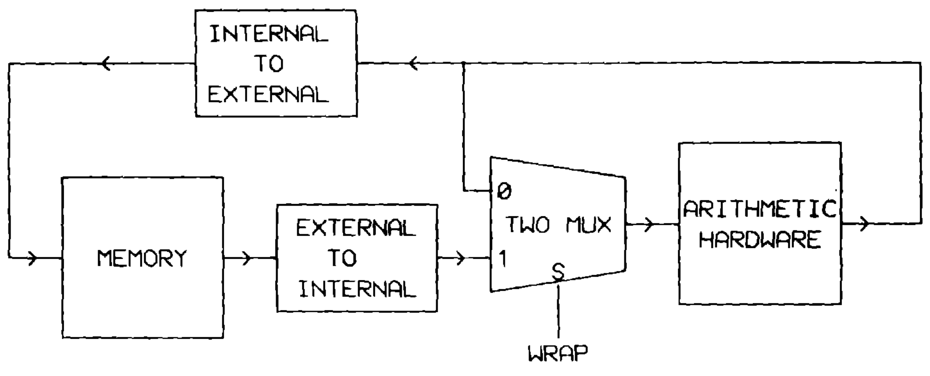


**Figure 4.2:** Block Diagram Of A System Which Uses Different Formats For Storage And Computing

## 4.2 Redundancy in the Internal Representation.

An example of a redundancy that is useful in the internal format occurs if the external format requires the use of floating-point *special symbols*. These are bit patterns which provide escapes from the normal meaning of the floating-point representation chosen. As an example, the S-1 architecture provides the special symbols **NAN** (not a number), **OVF** and **MOVF** (plus and minus overflow), and **UNF** and **MUNF** (plus and minus underflow). Actually, floating-point zero is implemented as a special symbol in the Mark IIA hardware, since otherwise a word with all zeros would be a very small positive number due to the hidden-bit fraction representation used in the S-1 architecture. (Since all floating-point numbers in the S-1 architecture have fractions whose magnitude is between 1 and 2, the bit just to the left of the binary point is redundant, and thus is left out of the architecturally defined formats.)

On the Mark IIA, these special symbols are encoded in a three-bit field in the internal format (see Figure 4.4). The value of the three bit field takes precedence over the value that the rest of the word would indicate; for instance, if the special symbol is zero then the number is zero, even if the fraction part of the word is nonzero. This simplifies many sections of hardware, since they don't have to worry about the hard-to-detect special cases — instead, the special cases can be propagated or generated separately. (There are actually two tags in the Mark IIA internal bus; the second tag is used only to represent the imaginary part of complex floating-point numbers.)

In fact, on the Mark IIA the propagation of special symbols is done with lookup tables. An example of such a table is shown in Figure 4.3. If a special symbol is generated by the table, then it will be forced into the special symbol field of the result; otherwise, the correct symbol is generated from the actual final result (including special symbols for overflow or underflow generated by this operation). Thus, the basic multiplication hardware need not worry about the special symbols, except to correctly detect newly generated error conditions.

**Multiplication (A*B)**

| A ↓ B→ | MOVF | -Y | MUNF | 0 | UNF | Y | OVF | NAN |
|---|---|---|---|---|---|---|---|---|
| MOVF | OVF | OVF | NAN | 0 | NAN | MOVF | MOVF | NAN |
| -X | OVF | X*Y | UNF | 0 | MUNF | -X*Y | MOVF | NAN |
| MUNF | NAN | UNF | UNF | 0 | MUNF | MUNF | NAN | NAN |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | NAN |
| UNF | NAN | MUNF | MUNF | 0 | UNF | UNF | NAN | NAN |
| X | MOVF | -X*Y | MUNF | 0 | UNF | X*Y | OVF | NAN |
| OVF | MOVF | MOVF | NAN | 0 | NAN | OVF | OVF | NAN |
| NAN | NAN | NAN | NAN | NAN | NAN | NAN | NAN | NAN |

**Figure 4.3:** Floating-Point Multiplication Propagation Table

## 4.3 Use of Extended Precision in the Internal Representation.

In addition to providing redundancy to speed up the latency of arithmetic operations, the internal format allows the use of extended precisions in a natural way.

As an example, the basic computation of a radix-two decimation-in-time FFT is the decimation-in-time butterfly, which consists of the computation [Rabiner75]:

```
procedure BUTTERFLY (C: complex; var A,B : complex);
        begin
        var T : complex;
        T ← A + C*B;
        B ← A - C*B;
        A ← T;
        end;
```

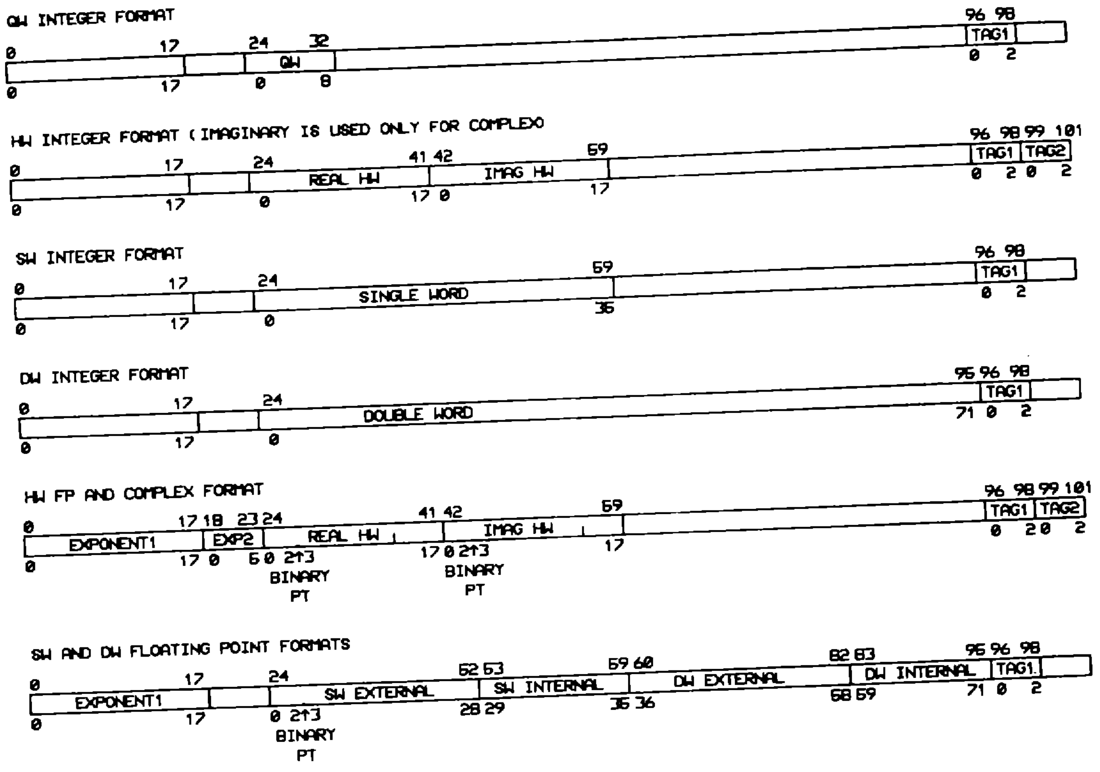


**Figure 4.4:** S-1 Mark IIA Internal Formats

If all of the operations of a butterfly are done in the internal format using extended precision, we can reduce the average error in the final results due to rounding by a factor of $\sqrt{3}$ (since two of the three rounding operations necessary for each result have been eliminated). Even greater reductions in the final rounding error could be achieved by using higher radix FFTs implemented in internal precision.

As another example of the value of an extended internal precision, consider the computation of a double-precision division, $A/B$, using multiplication as the iteration operator (i.e., a Newton iteration):

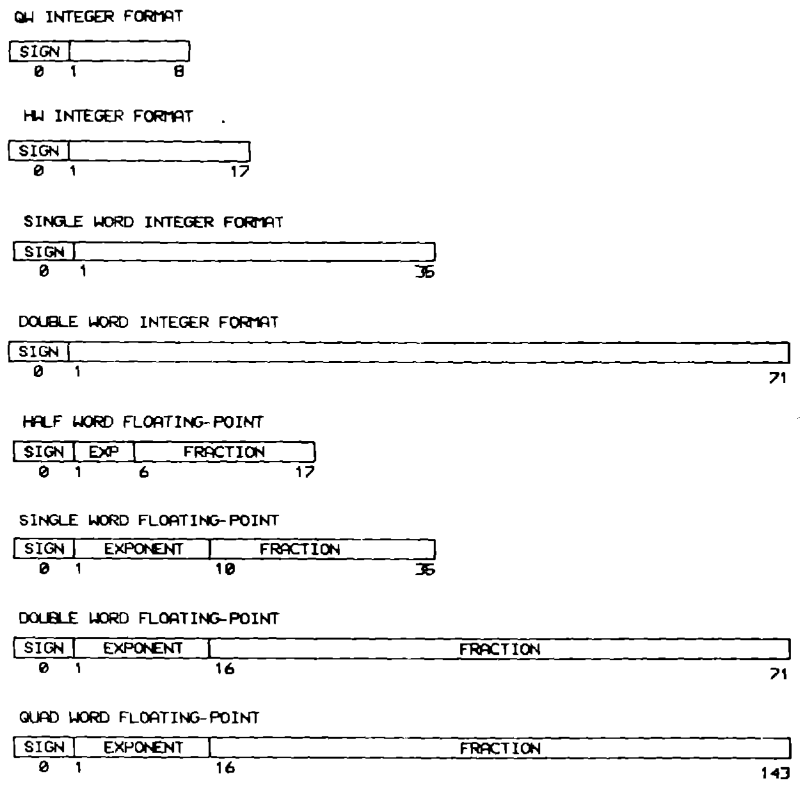


**Figure 4.5: S-1 Mark IIA External (Architectural) Formats**

```
procedure DoublePrecisionDivide (A,B : fraction);
        begin
        var Y : fraction;
        Y ← SinglePrecisionReciprocate(B);
        DoublePrecisionDivide ← A * Y * (2-Y*B);
        end;
```

Here we are computing an approximation, $Y$, to $1/B$ of slightly over half the final precision of the result. If $YB = 1 + \epsilon$ then $YB(2 - YB) = 1 - \epsilon^2$ and so $Y(2 - YB) = (1 - \epsilon^2)/B$. Thus if $Y = (1 + \epsilon)/B$ is half as accurate as necessary, $Y(2 - YB) = (1 - \epsilon^2)/B$ is accurate enough. However, in order to produce a final result with essentially full accuracy, we need to do the computations with more accuracy than is available in the final result (slightly more work is necessary to round perfectly, i.e., with an error of one-half the least significant bit). We could compute in "quad" precision (if such is available), but that would be very wasteful (of time), since we really only need a few extra bits of precision. The use of the internal format (rather than defining a new architectural format) solves this problem.

## 4.4 A Caution in the Use of the Internal Format.

To minimize latency of operations, it is desirable to wrap using the internal format when dependencies are detected. There is a temptation to wrap using the full internal precision in such a case. Unfortunately, this leads to the following problem:

The code fragment

```
X := X*Y;
SUM1 := SUM + X;
...
if (SUM1 ≠ SUM + X) then print("Terrible error!");
```

could be compiled into the S-1 assembly sequence:

```
FMULT X=X*Y
FADD SUM1=SUM+X
...
FADD RTA=SUM+X
SKP.NEQ SUM1,RTA,<Print TERRIBLE ERROR>
```

If the Mark IIA did wrap-around with the full internal precision, the terrible error could happen since the value of "X" used in the statement "SUM1 := SUM + X" is internal precision, while the value of "X" in "SUM1 ≠ SUM + X" is in external precision and hence the sums could differ. This is clearly undesirable, and so the hardware must be careful to limit the precision and range of results being wrapped as a result of dependencies in the instruction stream.

## 4.5 Drawbacks in the Use of an Internal Format.

Many computers use data formats designed such that many integer and floating-point operations (such as comparisons and moves) can be performed with the same instructions. Since the internal formats of integer and floating-point data will, in general, not be identical, seemingly unneccessary duplicate instructions must be provided so that the correct internal formats are generated (so that the correct wrapping is possible for both floating-point and integers). The cost of such "extra" instructions is low, and furthermore, when special symbols are provided for floating-point, such separate instructions are necessary in any case.

# 5. An Improved Floating-Point Addition Algorithm

The latency of floating-point addition is of critical importance to a high-speed "number-crunching" computer. For instance, with limited resources many linear recurrences take time which is proportional to the latency of an addition. For example, the sparse $k^{th}$-order linear recurrence

$$x_i = a_i x_{i-1} + b_i x_{i-k} + c_i,$$

needs approximately $k^2$ worth of resources (in this case, the ratio of addition rate to addition latency) to avoid being limited by the addition latency time. For many nonlinear recurrences, there is no obvious parallelism and so the performance of such recurrences is strictly limited by the latency. Also a large ratio between the pipeline rate and the latency puts a burden on the programmer to invent better algorithms and the compiler to produce better code, in order to raise computing efficiency to near its potential.

For these reasons, it is very important to implement a floating-point adder with mimimal latency that still has high pipelined throughput. This chapter outlines a method for decreasing the latency of normalized binary floating-point addition without affecting the pipeline rate.

To describe the new algorithm, it is useful to first recapitulate the usual method for computing floating-point additions. We will define a floating-point number as a pair $(e, f)$, where $f$ is a $p$-bit fixed-point fraction of magnitude between 1 and 2, i.e., we have

$$1 \leq |f| \leq 2.$$

The second number of the pair, $e$, is an integer. The number represented by this pair is $2^e \times f$.

## 5.1 Current Floating-Point Addition Algorithms.

Given two floating-point numbers, $A = (e_A, f_A)$ and $B = (e_B, f_B)$, we first describe the standard method to compute the normalized floating-point sum [Knuth81, Thornton70, Campbell62, Anderson67a, Stephenson75]. Figure 5.1 shows a block diagram of a possible hardware realization of this algorithm. This algorithm (or trivial variants thereof) is used on the Cray-1, CDC 7600, and the TI ASC, among others.

```
1. If $e_A < e_B$ set $r_A \leftarrow e_B - e_A$, $r_B \leftarrow 0$, and $e_{max} \leftarrow e_B$;
   if $e_A \geq e_B$ set $r_A \leftarrow 0$, $r_B \leftarrow e_A - e_B$, and $e_{max} \leftarrow e_A$.

2. Set $f_{tmp} \leftarrow f_A/2^{r_A} + f_B/2^{r_B}$

3. Set $S \leftarrow \lfloor \log_2(f_{tmp}) \rfloor$

4. Set $f_{result} \leftarrow f_{tmp}\, 2^{-S}$ and $e_{result} \leftarrow e_{max} + S$
```

**Algorithm 5.1:** Normal Floating-Point Addition

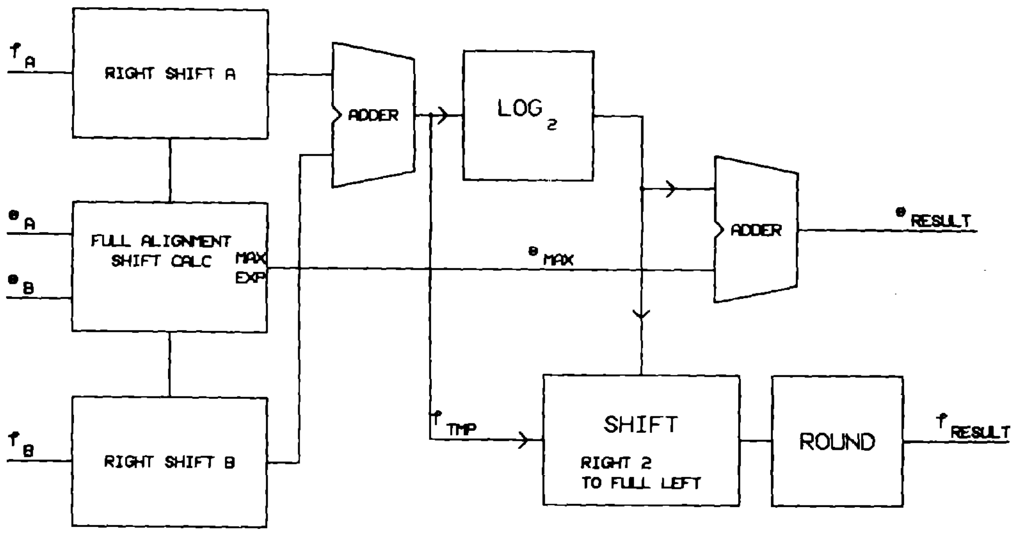


**Figure 5.1:** Old Floating-Point Addition Hardware

We are ignoring the details of representation, such as whether signed magnitude or twos-complement arithmetic is to be done and the size of the field which contains the exponent, $e$. These details do not affect the basic ideas of the algorithm.

## 5.2 A New Floating-Point Addition Algorithm.

We first show that the computation of $(f_A, e_A) + (f_B, e_B)$ can be resolved into one of two cases, each of which can be performed faster than can the previous algorithm.

**Case 1.** $|e_1 - e_2| \leq 1$ (i.e. $e_1 - e_2 = -1, 0$ or $1$) :

The shift used to implement the division by $2^{|e_1 - e_2|}$ is very small, in fact is either a zero or one place right shift. Furthermore, the amount of shift and the larger exponent (given that the assumption of case 1 is valid) can be determined by looking at the low two bits of both exponents. This case can be implemented by the sequence

```
1. If $e_A \equiv e_B$ (mod 4) set $R_A \leftarrow 0$, $R_B \leftarrow 0$, and $e_{max} \leftarrow e_A$;
   If $e_A \equiv e_B + 1$ (mod 4) set $R_A \leftarrow 0$, $R_B \leftarrow 1$, and $e_{max} \leftarrow e_A$;
   If $e_A \equiv e_B + 3$ (mod 4) set $R_A \leftarrow 1$, $R_B \leftarrow 0$, and $e_{max} \leftarrow e_B$;
   and we do not care about the case $e_A \equiv e_B + 2$ (mod 4).

2. Set $f_{tmp} \leftarrow f_A/2^{R_A} + f_B/2^{R_B}$.

3. Set S $\leftarrow \lfloor \log_2(f_{tmp}) \rfloor$

4. Set $f_{result} \leftarrow f_{tmp}\ 2^{-S}$ and
   $e_{result} \leftarrow e_{max}$ + S
```

**Algorithm 5.2:** Full Postnormalize Shift

**Case 2.** $|e_A - e_B| \geq 2$ :

In this case we need an arbitrarily large prealignment right shift. But at worst, we need to postnormalize by one place. To prove this, observe that

$$\begin{aligned}\left|f_{max} + f_{min}/2^{(e_{max}-e_{min})}\right| &\geq \left||f_{max}| - \left|f_{min}/2^{(e_{max}-e_{min})}\right|\right| \\ &\geq 1 - 2/2^{(e_{max}-e_{min})} \\ &\geq 1 - 1/2 \\ &\geq 1/2,\end{aligned}$$

where $e_{min}$ and $e_{max}$ are the same as in Algorithm 5.1, and $f_{min}$ and $f_{max}$ are the corresponding $f$'s. Thus, the result of the addition can never be so small as to need more than a one-place left shift. Also the sum can be never be larger than four and thus requires at most a one-place right shift. This case can be performed by the sequence

```
1. If $e_A < e_B$ set $R_A \leftarrow e_B - e_A$, $R_B \leftarrow 0$, and $e_{max} \leftarrow e_B$;
   if $e_A \geq e_B$ set $R_A \leftarrow 0$, $R_B \leftarrow e_A - e_B$, and $e_{max} \leftarrow e_A$.
2. Set $f_{tmp} \leftarrow f_A/2^{R_A} + f_B/2^{R_B}$.
3. If $f_{tmp} \geq 2$ set $f_{result} \leftarrow f_{tmp}/2$ and $e_{result} \leftarrow e_{max} + 1$,
   If $2 > f_{tmp} \geq 1$ set $f_{result} \leftarrow f_{tmp}$ and $e_{result} \leftarrow e_{max}$,
   If $f_{tmp} < 1$ set $f_{result} \leftarrow 2f_{tmp}$ and $e_{result} \leftarrow e_{max} - 1$,
```

**Algorithm 5.3:** Full Alignment Shift

Since these two cases are mutually exclusive, we can build hardware to execute both simultaneously (see Figure 5.2). Thus, on any given execution path, there is one big shift and one small shift; this compares to two big shifts required for the normal algorithm. Furthermore, the "Full Postnormalize Shift" case in general has the longest delay, and the fact that only the low two bits of both $e_A$ and $e_B$ need to be examined improves the latency of this path.

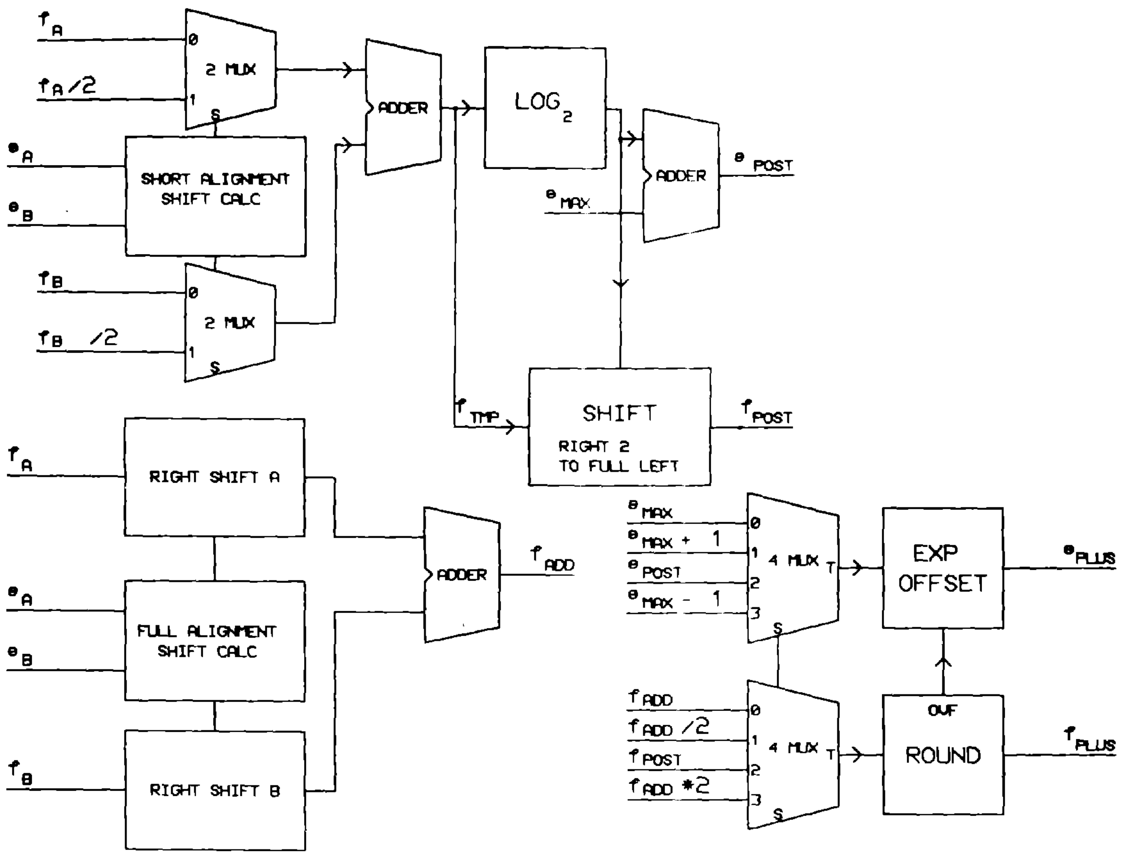


**Figure 5.2:** New Floating-Point Addition Hardware

The fact that floating-point addition can be classified into two mutually exclusive cases as described above seems to have been part of computer folklore for some time, although no one seems to have appreciated its value for a hardware implementation. The first mention of it that the author has been able to find is [Sweeney65], who used it to explain statistical observations of floating-point addition shift behaviour. W. Kahan has related that the idea has been used in software written by his group at Berkeley [personal communication] and [Field69] also pointed out its use in software floating-point implementations. Of course, many hardware implementations of floating-point addition which spend variable amounts of time shifting have benefitted (perhaps unwittingly) from this fact.

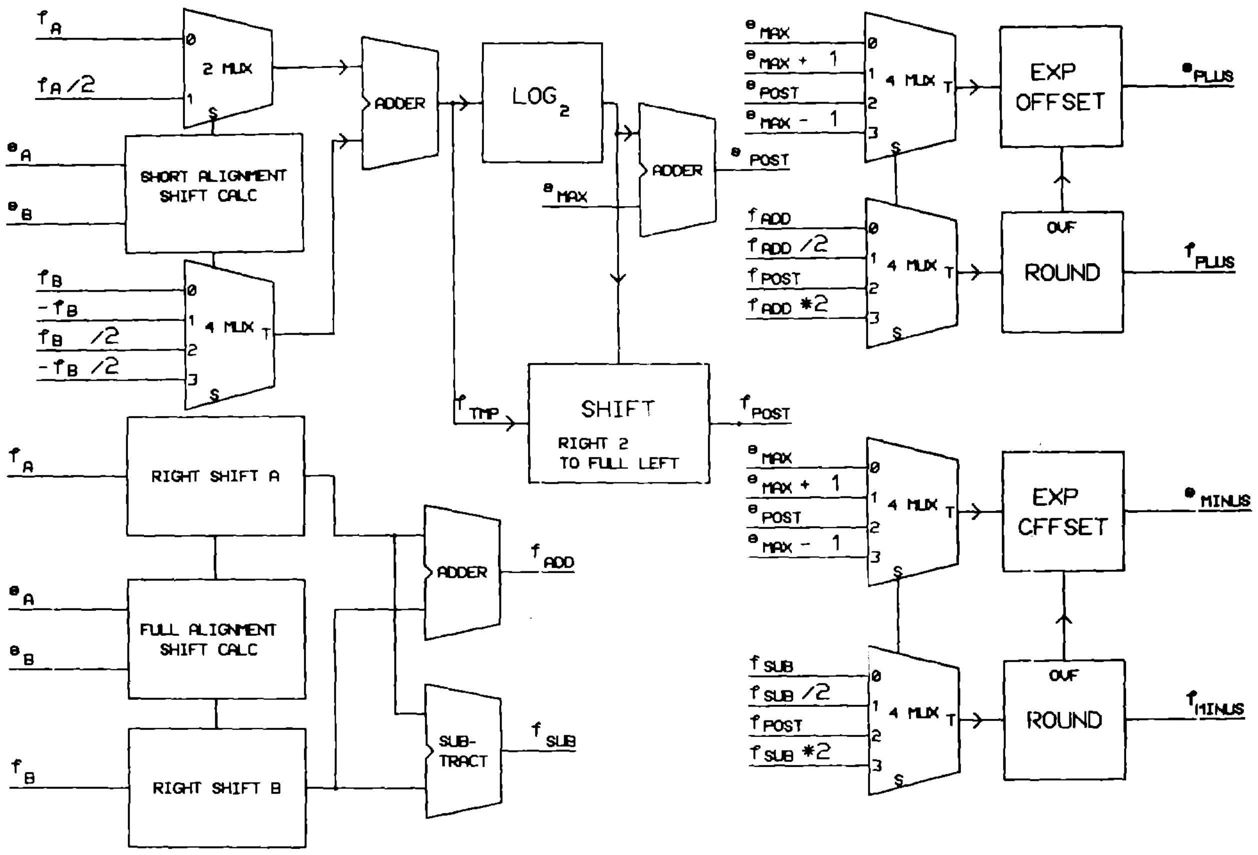


**Figure 5.3:** Simultaneous Sums and Differences

## 5.3 Simultaneous Floating-Point Adds and Subtracts.

Many algorithms (such as the FFT) require the generation of both $A+B$ and $A-B$. The algorithm given in this paper may be extended to allow the simultaneous calculation of the floating-point sum and difference for much less than twice the hardware cost of either. First, note that in case 1 above a full postnormalization shift is needed only if the signs of $f_A$ and $f_B$ are different. Now if we add a "complementor" to the $B$-leg input of the "Full Postnormalize Shift" path and complement $B$ if $f_A$ and $f_B$ had the same sign, we can ensure that if either the $A+B$ or $A-B$ required a full postnormization shift, it indeed will be done. The "Full Alignment Shift" path must be partially duplicated in that separate data paths are needed after the shifters (see Figure 5.3). Since the "Full Postnormalize Shift" circuit both contains more logic and is the critical path, this is of minor importance (in that it has a small effect on the overall logic count and no effect on the circuit speed).

# 6. Evaluation Of Elementary Functions

Consider the well-known [Flynn70, Rabinowitz61, Shaham72] technique for evaluating $1/z$ by generating an initial approximation, $a_0$, for $1/z$ using a table-lookup on the high bits of $z$, and then using the iteration $a_{i+1} = a_i(2 - a_i z)$ until the desired accuracy is reached. If we let $x$ be the high bits of $z$ used for the table-lookup and let $y = z - x$, then the combination of the table-lookup and the first iteration is equivalent to expanding the Taylor series for $1/z = 1/(x + y)$ about the initial point of approximation, $x$, i.e.

$$a_1 = a_0(2 - a_0 z) = \frac{1}{x} - \frac{y}{x^2},$$

where $a_0 = 1/x$ is the initial approximation. Furthermore, if we compute $1/x^2$ by using table-lookup on $x$ (which can be done in parallel with the table-lookup of $1/x$), with one multiplication ($y(1/x^2)$) we have doubled the precision of our approximation ($1/x$). We shall show that use of further terms of the Taylor series leads to a fast method for evaluating many functions, whereby in one multiplication time (during which two parallel multiplications occur), we can triple the precision of the table-lookup approximation.

## 6.1 Mathematics of the Approximation Technique.

We wish to compute a suitable function, $f(z)$, to $p$-bits of accuracy. More precisely, over some limited range of the argument $z$, we would like to compute an approximation $\hat{f}(z)$, such that $|f(z) - \hat{f}(z)| \leq 1/2^{p+1}$ for all $z$ in the restricted range.

For simplicity, assume that $0 \leq z < 1$ is the range in which we are interested. Thus $z$ is a fixed-point fraction with $p_z$ bits. We will break $z$ into two pieces, $x$ and $y$, where $x = \lfloor z2^{p_x} \rfloor / 2^{p_x}$ and $y = (z - x)2^{p_x}$, so that $z = x + y/2^{p_x}$. It should be clear that $0 \leq x < 1$ and $0 \leq y < 1$ and that $x$ and $y$ are fixed point fractions with $p_x$ and $p_z - p_x$ bits, respectively. We are going to make a number of approximations to $x$ and $y$ which use various numbers of the leading bits of each. We will use the notation $p_{x_i}$ and $p_{y_i}$ for the number of bits of $x$ or $y$ used in the $i^{th}$ term of the approximation.

We have (using the Taylor expansion for $f$ at $x$)

$$\begin{aligned} f(z) = f(x + \frac{y}{2^{p_x}}) & \\ = \quad & f(x) && (1a)\\ & + \frac{y}{2^{p_x}} f'(x) && (1b)\\ & + (\frac{y}{2^{p_x}})^2 f''(x)/2! && (1c)\\ & + (\frac{y}{2^{p_x}})^3 f'''(x)/3! && (1d)\\ & + (\frac{y}{2^{p_x}})^4 f''''(x+\beta)/4!, && (1e) \end{aligned}$$

where $0 \leq \beta \leq y/2^{p_x}$.

We will analyze the approximation to $f$ of (see Figure 6.1)

$$\hat{f}(z) = \mathrm{round}(f(x), p+6) \tag{2a}$$

$$+ \frac{y}{2^{p_x}} \mathrm{round}(f'(x), p+6-p_x) \tag{2b}$$

$$+ \mathrm{round}\left(\left(\frac{\mathrm{truncr}(y, p_{y_2})}{2^{p_x}}\right)^2, p+6\right) \times \mathrm{round}\left(\frac{f''(x)}{2!}, p+7-2p_x\right) \tag{2c}$$

$$+ \mathrm{round}\left(R\left(\mathrm{truncr}(x, p_{x_3}), \frac{\mathrm{truncr}(y, p_{y_3})}{2^{p_x}}\right), p+7-3p_x\right) \tag{2d}$$

where

$$\mathrm{round}(z, q) = \lfloor z2^q + 1/2 \rfloor / 2^q,$$

(i.e., normal rounding to $q$-bits)

$$\mathrm{truncr}(z, q) = (\lfloor z2^q \rfloor + 1/2)/2^q$$

(also known as von Neumann rounding or jamming) and

$$R(u, v) = (v/2^{p_x})^3 f'''(u)/3! + (v/2^{p_x})^4 f''''(u)/4!.$$

Of these four terms constituting the desired approximation, note that (2a) and (2d) consist of table-lookups, (2b) of a table-lookup followed by a multiplication, and (2c) of two table-lookups followed by a multiplication. Note also that all the table-lookups may be performed in parallel, as may the two multiplications.

Let us assume that the general term of the Taylor series for $f$ satisfies

$$\left| f^{(n)}(z) \right| \leq n!$$

for $0 \le z < 1$ and $n \ge 0$. In addition, let us choose $p_x$ such that it is technologically feasible to perform fast table lookups on $x$ (e.g., about 10-16 bits, using 1981 technology). The values of the terms $f(x)$, $f'(x)$ and $f''(x)$ thus can be obtained directly from ROM or RAM. (This implies that $p_{x_0} = p_{x_1} = p_{x_2} = p_x$.) Also note that $p_{y_1} = p_y = p_z - p_x$, since we use the full precision of $y$ in term (2b). This is not strictly necessary, since we could approximate $y$ with enough of its leading bits (at least $p + 7 - p_x$) to make the error of truncation small.

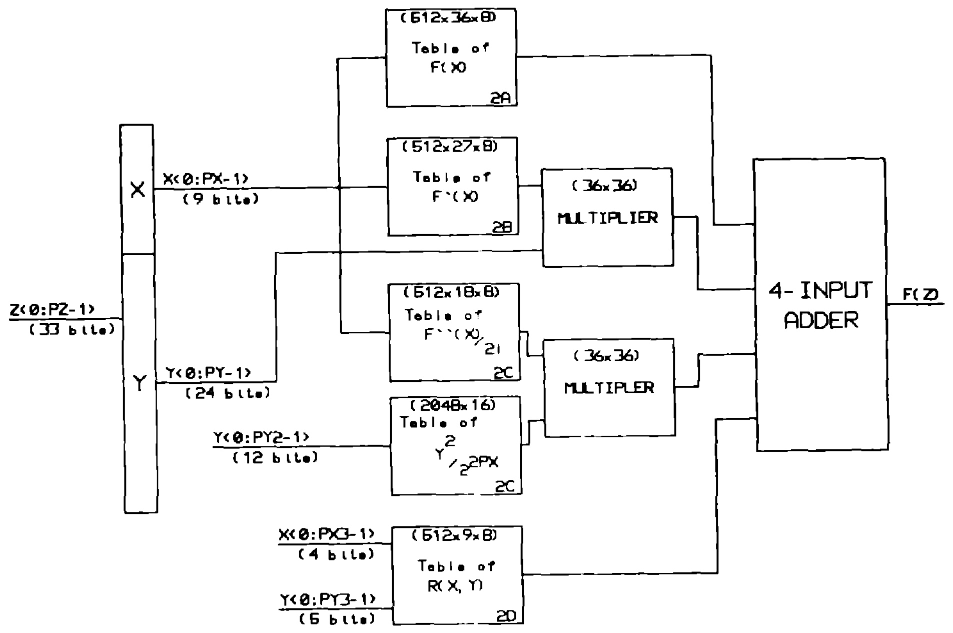


**Figure 6.1:** Block Diagram of Taylor Series Approximation Hardware

The term $(y/2^{p_x})^2$ can be obtained by table lookup on the high $p_{y_2}$ bits of $y$, if $2p_x + p_{y_2} \geq p + 2$. To see this, note that the approximation used in the table lookup is

$$y \approx \frac{\lfloor y2^{p_{y_2}} \rfloor + \frac{1}{2}}{2^{p_{y_2}}} = y + \frac{a}{2^{p_{y_2}}},$$

where $-1/2 < a \leq 1/2$. The error involved in using this to approximate $(y/2^{p_x})^2$ is

$$\begin{aligned} \left| \left(\frac{y}{2^{p_x}}\right)^2 - \left(\frac{y + \frac{a}{2^{p_{y_2}}}}{2^{p_x}}\right)^2 \right| &= \frac{1}{2^{2p_x}} \left| \frac{2ay}{2^{p_{y_2}}} - \frac{a^2}{2^{2p_{y_2}}} \right| \\ &\leq \frac{1}{2^{2p_x + p_{y_2}}} + \frac{1}{2^{2p_x + 2p_{y_2} + 2}}. \end{aligned}$$

Term (2d) is found by table lookup on the high bits of $x$ and $y$. We will use the high $p_{x_3}$ and $p_{y_3}$ bits of x and y. Using the same approach as before, we approximate $x \approx x + \tau/2^{p_{x_3}}$ and $y \approx y + \lambda/2^{p_{y_3}}$, where $-1/2 < \tau \leq 1/2$ and $-1/2 < \lambda \leq 1/2$. Then the error in this approximation to term (2d) will be (leaving out intermediate steps)

$$\begin{aligned} &\left| R(x,y) - \left(\frac{y + \frac{\lambda}{2^{p_{y_3}}}}{2^{p_x}}\right)^3 \frac{f'''(x + \frac{\tau}{2^{p_{x_3}}})}{3!} \right. \\ &\qquad\qquad \left. - \left(\frac{y + \frac{\lambda}{2^{p_{y_3}}}}{2^{p_x}}\right)^4 \frac{f''''(x + \frac{\tau}{2^{p_{x_3}}})}{4!} \right| \\ &\leq \frac{2}{2^{3p_x + p_{x_3}}} + \frac{3/2}{2^{3p_x + p_{y_3}}} \\ &\qquad + O\left(\frac{1}{2^{4p_x + \min(p_{x_3}, p_{y_3})}}\right) + O\left(\frac{1}{2^{3p_x + 2\min(p_{x_3}, p_{y_3})}}\right). \end{aligned}$$

We now have all of the pieces to find the total error in our approximation to (1). The five rounding errors sum to at most $1/2^{p+5}$. Thus the error in (2c) can be made close to $1/2^{p+2}$ by making $2p_x + p_{y_2} \geq p+2$. We can make the error in (2d) be less than $7/2^{p+5}$ if $3p_x + p_{x_3} \geq p+4$ and $3p_x + p_{y_3} \geq p+4$. Furthermore, we ignore a number of smaller error terms because they make little or essentially no contribution to the total error. Summing these errors gives a total maximum error less than $1/2^{p+1}$, as desired. Note that this is the maximum error — the average error will tend to be much less, due to cancellation effects.

Summarizing the foregoing assumptions, we have

$$\left|f^{(n)}(z)\right| \leq n!, \tag{3a}$$

$$2p_x + p_{y_2} \geq p+2, \tag{3b}$$

$$3p_x + p_{y_3} \geq p+4, \tag{3c}$$

$$3p_x + p_{x_3} \geq p+4, \tag{3d}$$

together with technology constraints on the choice of $p_x$, $p_{y_2}$, $p_{x_3}$ and $p_{y_3}$.

We shall now select $p_x$, $p_{y_2}$, $p_{x_3}$ and $p_{y_3}$ to minimize the total number of bits in the tables. There are five tables used, and the sum of their sizes (in order of their appearance in (2)) is

$$2^{p_x}(p+6) + 2^{p_x}(p+6-p_x) + 2^{p_{y_2}}(p+6-2p_x) \\ + 2^{p_x}(p+7-2p_x) + 2^{p_{x_3}+p_{y_3}}(p+7-3p_x).$$

Clearly, we can let $p_{y_2} = p+2-2p_x$ and $p_{x_3} = p_{y_3} = p+5-3p_x$, since these are the minimum values which satisfy the constraints. Thus we must minimize

$$2^{p_x}(3p+19-3p_x) + 2^{p+2-2p_x}(p+6-2p_x) \\ + 2^{2p+10-6p_x}(p+7-2p_x)$$

as a function of $p_x$. Without proof, we state the result that the minimum occurs at $p_x \approx p/3$, which satisfies the constraints if $p_x \geq 3$. The table involving only bits of $y$ need not be duplicated when using this technique for evaluating more than one function; this drives the optimum value of $p_x$ even lower, since the $y^2$ table is common to all of the functions. The total table size (in bits) at $p_x = p/3$ is approximately

$$2^{\frac{p}{3}}(3p + 45) + 7000$$

or if $k$ functions are implemented:

$$2^{\frac{p}{3}}((2k + 1)p + 20k + 25) + 7000k.$$

## 6.2 Some Functions Which Satisfy The Constraints.

Table 6.1 lists some common functions which satisfy the necessary conditions developed above. The examples assume a radix-2 floating-point format. Since the dynamic range of the result must be limited to maintain relative accuracy (the dynamic range is limited essentially to 2), we are forced in some cases to compute another function which can easily be transformed to the correct result. In fact, the correct way to regard this method is to consider it a technique for computing a fixed-point function of a fixed-point number. To maintain relative precision in floating-point may involve additional work. This is why we compute $\sin(z)/z$ instead of $\sin(z)$ (since sin has a zero at zero). Also note that $1 - \mathrm{erf}(z) \leq \pi e^{-z^2}/(2z)$ for $z \geq 1$. Thus for $z > 8$, we have $\mathrm{erf}(z) = 1$ to within $2^{-92}$ or $10^{-28}$. Also, the derivatives of $\operatorname{erf}(z)$ are so small when $z > 1$ that much coarser tables can be used than for the range $0 \le z < 1$. The tables used in computing $2/z$ and $\sqrt{z}$ are double-sized, since in computing $2/z$ we need to look at the sign of $z$ and in computing $\sqrt{z}$ we must examine the low exponent bit, i.e., we use different approximations, depending upon whether the exponent is

| <u>Base function</u> | <u>Domain</u> | <u>Example of use</u> |
|---|---|---|
| $\sin(\frac{\pi z}{2})/z$ | $0 \le z < 1$ | $\sin(z) = \begin{cases} i f(i), & (0 \le j < 1) \\ (1-i) f(1-i), & (1 \le j < 2) \\ -i f(i), & (2 \le j < 3) \\ -(1-i) f(1-i), & (3 \le j < 4) \end{cases}$ where $i = \frac{2z}{\pi} \bmod 1$ and $j = \frac{2z}{\pi} \bmod 4$ |
| $\log_2(z)$ | $1 \le z < 2$ | $\log_2(2^e m) = e + f(m)$ |
| $2/z$ | $1 \le \lvert z\rvert < 2$ | $1/2^e m = 2^{-e-1} f(m)$ |
| $\sqrt{z}$ | $1 \le z < 4$ | $\sqrt{2^{2e_1+e_2} m} = 2^{e_1} f(2^{e_2} m)$ |
| $2^z$ | $0 \le z < 1$ | $2^z = 2^{\lfloor z \rfloor} f(z - \lfloor z \rfloor)$ |
| $\arctan(z)/z$ | $0 \le z < 1$ | $\arctan z = \begin{cases} f(\frac{1}{z})/z - \frac{\pi}{2}, & (z < -1) \\ -z f(z), & (-1 \le z < 0) \\ z f(z), & (0 \le z < 1) \\ \frac{\pi}{2} - f(\frac{1}{z})/z, & (1 \le z) \end{cases}$ |
| $\operatorname{erf}(z)/z$ | $0 \le z < 8$ | $\operatorname{erf} z = \begin{cases} -1, & (z < -8) \\ -z f(z), & (-8 \le z < 0) \\ z f(z), & (0 \le z < 8) \\ 1, & (8 \le z) \end{cases}$ |

Table 6.1

odd or even [Fike68].

## 6.3 Practical Implementation.

The technique just described has been implemented in the S-1 Mark IIA processor [S-1 Project 79] to evaluate elementary functions in single-precision floating-point (which has a sign bit, 9 exponent bits, a hidden fraction bit, and 26 bits of fraction). The numbers in parenthesis (in Figure 6.1) are the actual sizes of the tables used. The Taylor series is actually evaluated to 29 bits of accuracy and then rounded to 27 bits. This implies that the functions satisfy an equation of the form $\hat{f}(x) = f(x)(1+\epsilon)$, where $\epsilon \leq 0.62/2^{27}$. All of the approximations used satisfy the correct monotonicity properties (partly as a consequence of evaluating to extra precision before rounding). In addition, some of the functions satisfy necessary special properties, such as $|\sin| \leq 1$. Evaluation to extra precision is also valuable for computing double-precision reciprocation and square-root. The S-1 double-precision floating-point format has 57 fraction bits. Since we have an approximation which has more than one-half of the desired precision, one Newton iteration will suffice to finish the approximation.

All of the table lookups are done in the first stage of the multiplier functional unit, with the normal operation of the succeeding stages of the multipier being delayed by 25 nanoseconds (to provide the time needed for the table look-ups). The multiplier has four 18 x 36 multipliers which can be reconfigured as two 36 x 36 multipliers. The subsequent pipe stages (accumulate, normalize, round and normalize) are shared by the floating-point multiply and elementary function evaluation functional units. The multiplier has a beginning-to-end latency (time duration from input to output) of either 125 or 150 nanoseconds, depending on how it is being used. This arrangement readily supports the pipelined evaluation of $1/z$, $\sqrt{z}$, $\log_2 z$, and $2^z$ at 25 nanoseconds per datum and $y/z$, $\ln z$, $\log z$, $e^z$, $\arctan z$, $\sin z$, $\cos z$ at 50 nanoseconds per datum.

## 6.4 Rounding.

There is a minor problem with this form of function evalution, in that it doesn't round "perfectly", i.e., it doesn't allow rounding of the nearest representable floating point number to the exact result. (Here "perfect" rounding means that the error in the result is less than or equal to one-half the least significant bit.) Of the functions discussed above, reciprocation and division (and occasionally square root) are the only ones for which serious attempts are ever made to do "perfect" rounding. The error due to the approximate method for elementary function evaluation presented here can be considered as additional rounding error, and can be made as small as desired by making $p$ as large as necessary before rounding. This type of error has been extensively studied by numerical analysts, and has been found to be quite acceptable where function evaluation speed is important. There is no a priori limit known to the precision to which one might have to evaluate sine, cosine, logarithm, exponential and arctangent in order to round correctly in these cases [Kahan81]. Of course, for a fixed size word, one could determine the precision needed to round each value ahead of time and so place a bound on the precision required to correctly to round any value. This is clearly not worth the small increment in precision that it actually yields. For division, reciprocation, and square root, it is interesting to note that in order to round "perfectly" to $p$ bits, it is sufficient to evaluate the result to $2p$ bits before rounding.

# 7. Parallel Transposition and Bit-Address-Reversing

Rapid execution of matrix transposition and bit-address-reversing are two important capabilities needed by a high-performance digital computer. Matrix transposition (or at least parallel access to both rows and columns of a matrix) must be done in the course of executing many linear equation algorithms, while bit-address-reversing is required to unscramble (or scramble) the locations of the data of the standard in-place Cooley-Tukey Fast Fourier Transform (FFT).

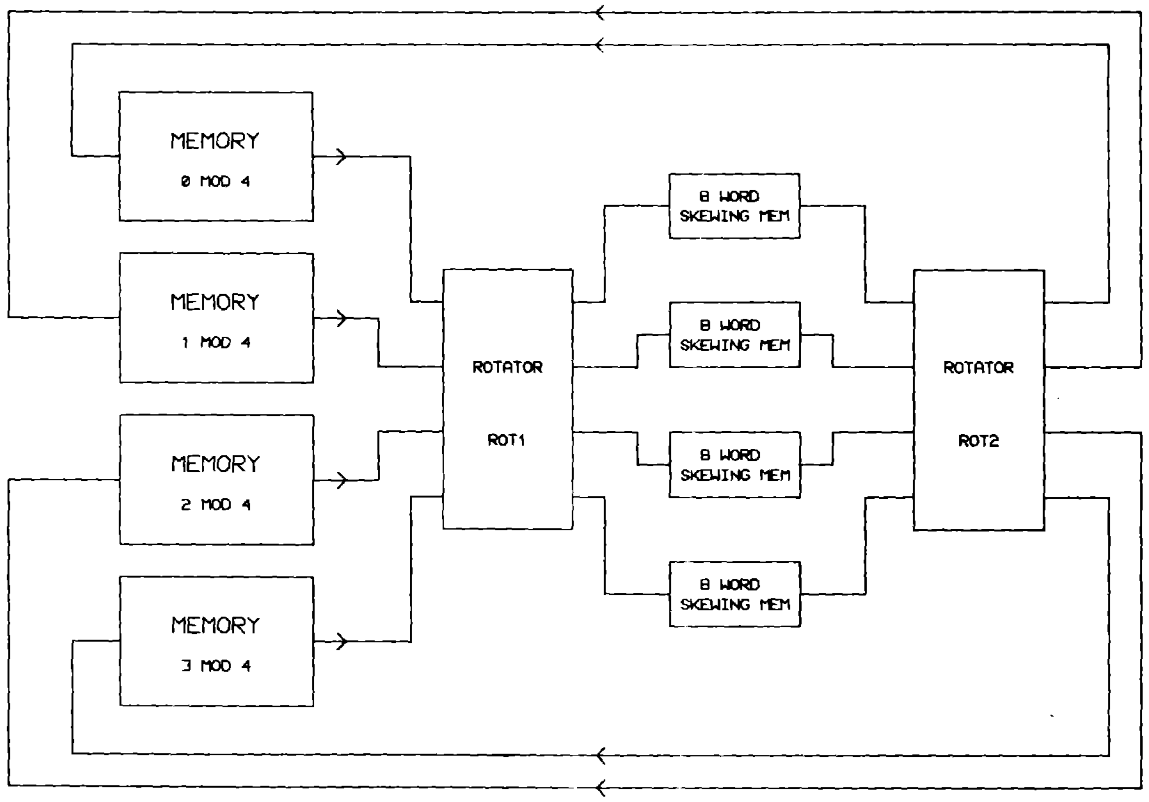
The use of memory interleaving to increase bandwidth is nearly universal on high-speed computing machines. However, with standard data representations of matrices, it is difficult to use such interleaving effectively to accomplish matrix transposition. Skewing techniques have been described [Budnik71,Kuck78,Lawrie75] to support the more efficient use of such interleaved memories, but these techniques have not enjoyed widespread acceptance by the computer-using community or by compiler writers.

Using data rotators and individually addressed small memories (all of the same size as the degree of interleaving), we show how to implement matrix transpose and bit-address-reverse, starting and ending with standard representations of matrices and arrays. The structure of this hardware is such that it can also quite usefully act as a queue for buffering data between memory and an execution unit (which, in fact, has been done on the S-1 Mark IIA).

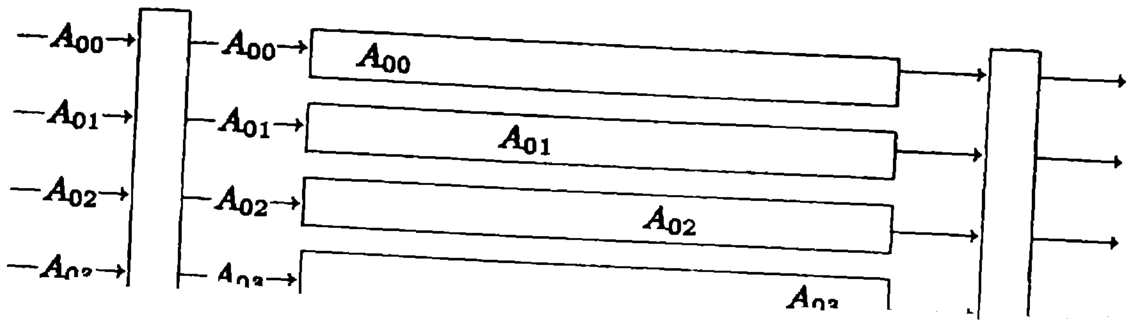
## 7.1 The Algorithm.

The basic idea is taken directly from data skewing methods. With normal storage representation and N-way interleaved memory, N words of any row can be read in parallel (assuming row major matrix storage). By reading $N$ such $N$-word sequences and storing them in the skewed representation in a special memory, it is then possible to read from the special memory $N$ words from any row or column. In particular, the implementation of transpose developed in this research consists of reading out in parallel the columns of the input (sub-)matrix and storing them back into memory as the rows of the output (sub-)matrix. A similar technique may be used to perform the bit-address-reversing required by the standard FFT algorithm.

To better explain how the method works, some examples may be useful. Figure 7.1 is the block diagram of the hardware implementation (for $N = 4$) which we will use in the examples.



[illegible] 1: Block Diagram Of Transpose Hardware



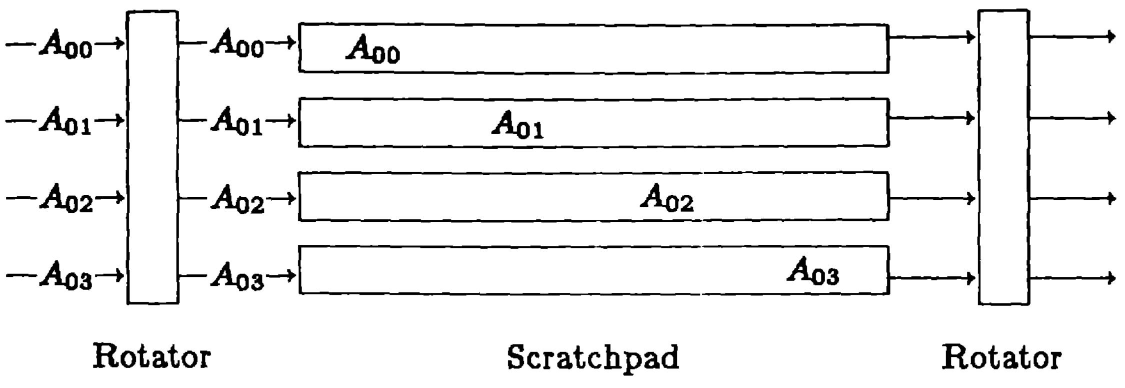


**Figure 7.2:** Reading The Subvector, Step 1

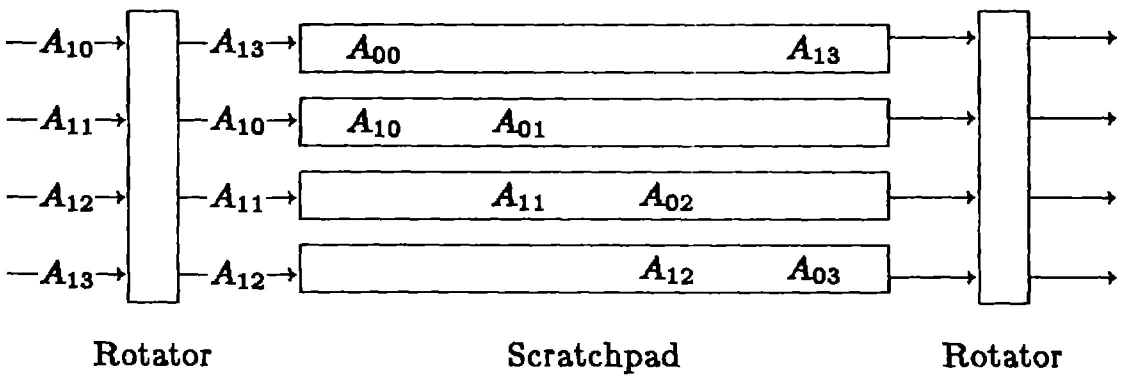


**Figure 7.3:** Reading The Subvector, Step 2

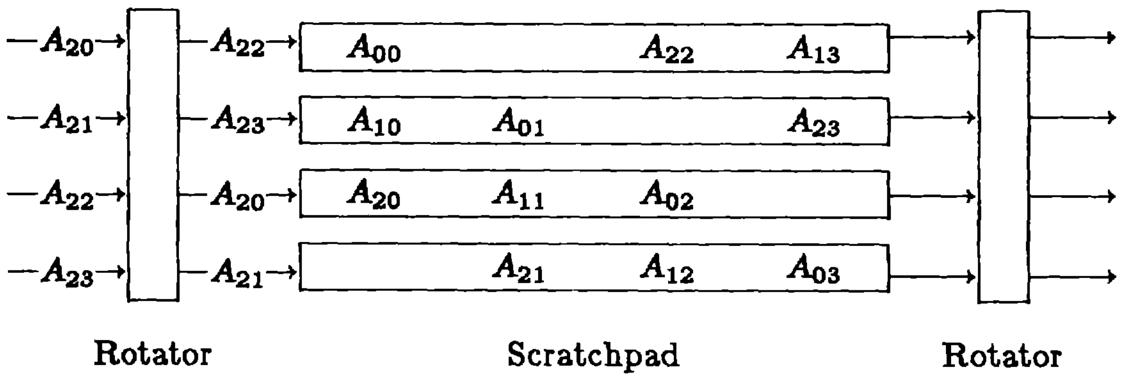

$A_{20}$
$A_{21}$
$A_{22}$
$A_{23}$
$A_{22}$
$A_{23}$
$A_{20}$
$A_{21}$
$A_{00}$ $A_{22}$ $A_{13}$
$A_{10}$ $A_{01}$ $A_{23}$
$A_{20}$ $A_{11}$ $A_{02}$
$A_{21}$ $A_{12}$ $A_{03}$
Rotator
Scratchpad
Rotator

**Figure 7.4:** Reading The Subvector, Step 3

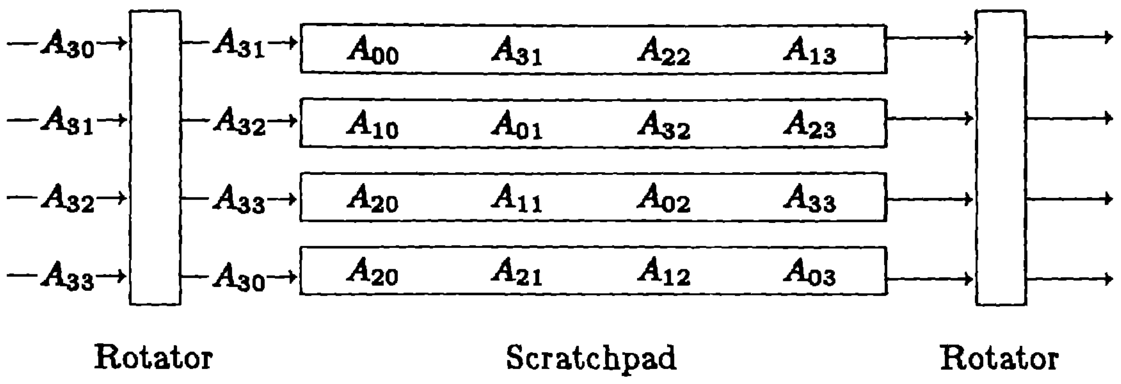

$A_{30}$
$A_{31}$
$A_{32}$
$A_{33}$
$A_{31}$
$A_{32}$
$A_{33}$
$A_{30}$
$A_{00}$ $A_{31}$ $A_{22}$ $A_{13}$
$A_{10}$ $A_{01}$ $A_{32}$ $A_{23}$
$A_{20}$ $A_{11}$ $A_{02}$ $A_{33}$
$A_{20}$ $A_{21}$ $A_{12}$ $A_{03}$
Rotator
Scratchpad
Rotator

**Figure 7.5:** Reading The Subvector, Step 4

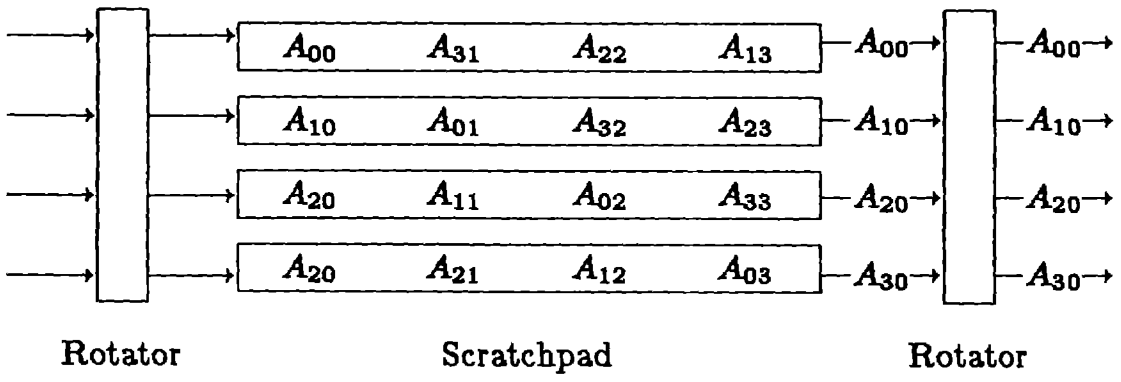

$A_{00}$ $A_{31}$ $A_{22}$ $A_{13}$
$A_{10}$ $A_{01}$ $A_{32}$ $A_{23}$
$A_{20}$ $A_{11}$ $A_{02}$ $A_{33}$
$A_{20}$ $A_{21}$ $A_{12}$ $A_{03}$
$A_{00}$
$A_{10}$
$A_{20}$
$A_{30}$
$A_{00}$
$A_{10}$
$A_{20}$
$A_{30}$
Rotator
Scratchpad
Rotator

**Figure 7.6:** Writing The Subvector, Step 1

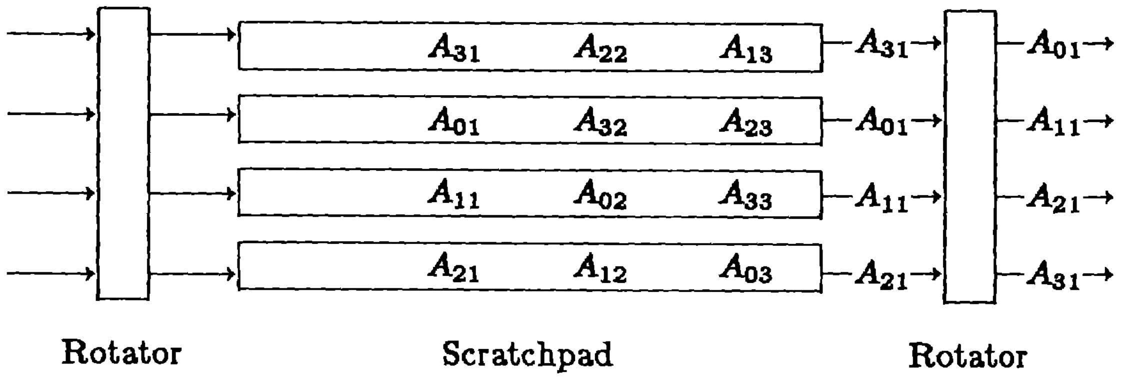

$A_{31}$ $A_{22}$ $A_{13}$
$A_{01}$ $A_{32}$ $A_{23}$
$A_{11}$ $A_{02}$ $A_{33}$
$A_{21}$ $A_{12}$ $A_{03}$
$A_{31}$
$A_{01}$
$A_{11}$
$A_{21}$
$A_{01}$
$A_{11}$
$A_{21}$
$A_{31}$
Rotator
Scratchpad
Rotator

**Figure 7.7:** Writing The Subvector, Step 2

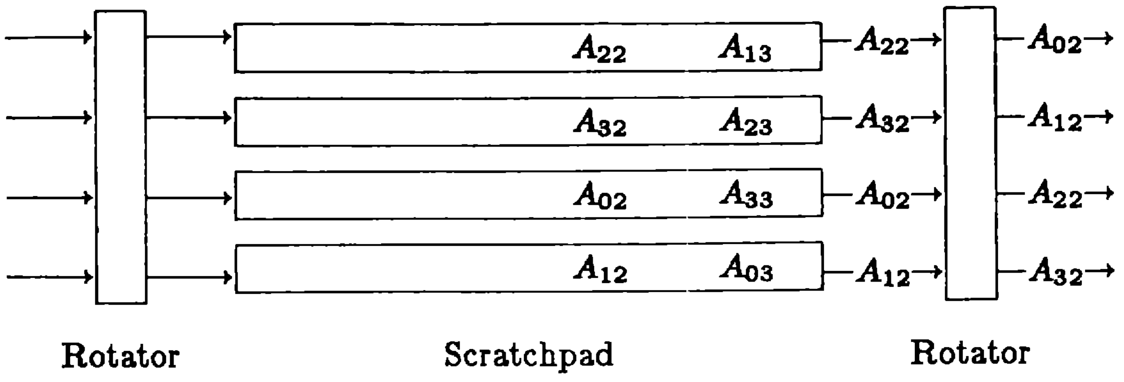


**Figure 7.8:** Writing The Subvector, Step 3

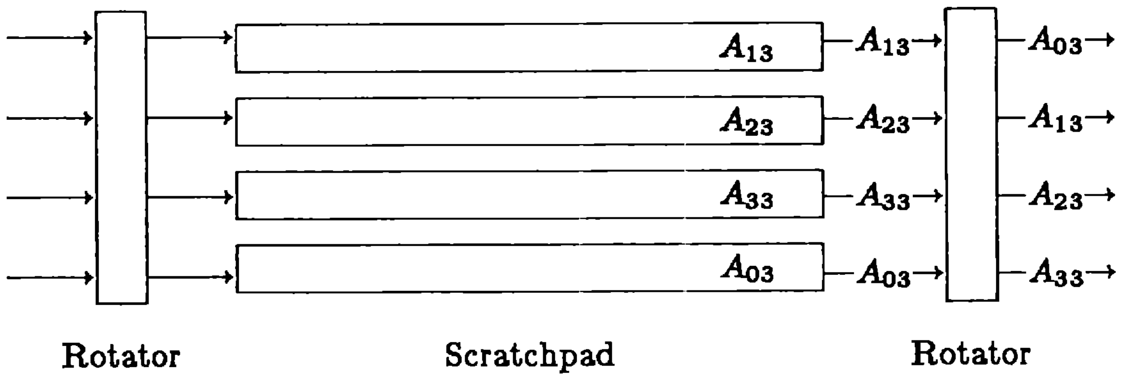


**Figure 7.9:** Writing The Subvector, Step 4

### 7.1.2 Example: Bit-Address-Reversing.

As a second example, suppose instead that we have an one-dimensional array $B_i$, $0 \leq i \leq 2^p - 1$, and that we would like to "bit-address-reverse" $B$, i.e. we would like form a new array, $C_i$, such that $C_i = B_{bitreverse(i,p)}$. (The function *bitreverse*$(i,p)$ reverses the binary digits of a $p$-bit number $i$, i.e., if $i = i_0 i_1 ... i_{p-1}$ is the binary representation of $i$, then the binary representation of *bitreverse*$(i,p)$ is $i_{p-1} ... i_1 i_0$. See [Polge74] or [Rabiner75] for a derivation of the need for bit-reveral in the computation of the Fast Fourier Transform.) Furthermore, we want to use the same memory locations for both $B$ and $C$, so that the transformation is done "in-place". On a computer with non-interleaved memory, this can be done in a straightforward fashion by a program of the form

```
for I := 0 step 1 until 2^N-1 do
        if I > BITREVERSE(I,N) then B[I] ↔ B[BITREVERSE(I,N)];
```

Here "↔" is the exchange operator and the function "BITREVERSE(I,N)" returns the bit reverse of the low $N$ bits of $I$. On a computer with interleaved memory, such a simple loop doesn't work well, since either the memory reads or writes can't be sequential, and thus the algorithm is unable to effectively use the parallelism of the memory.

Let us resolve a subscript of the array, $B_{iXj}$, into a concatenation of three fields: $i$ and $j$ are two-bit fields (for $N = 4$; in general, they would be $k$ bit fields for $N = 2^k$), and $X$ is the middle $p - 4$ bits of the subscript. Consider all possible array elements whose middle bits of the subscript are $X$ (i.e., $B_{iXj}$, $0 \leq i \leq 3$ and $0 \leq i \leq 3$) and those array elements whose middle bits of the subscript are $Y = bitreverse(X)$. Then, the desired transformation is a one-to-one mapping between these two sets. Thus we can replace the single element exchange in the simple loop above with a set exchange, where the sets are read and written in parallel. Figure 7.10 through Figure 7.17 illustrate the fetch and bit-address-reveral of $B_{iXj}$. It is clear how the same operation for $B_{iYj}$ can be overlapped so that the store of the bit-address-reversed $B_{iXj}$ can be made directly into the locations just read during the bit-address-reversal of $B_{iYj}$.

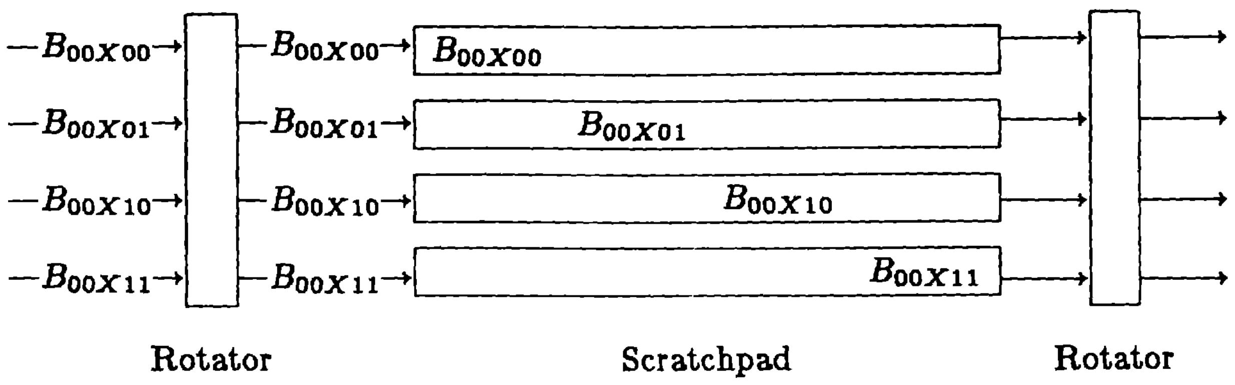


**Figure 7.10:** Reading The Subvector, Step 1

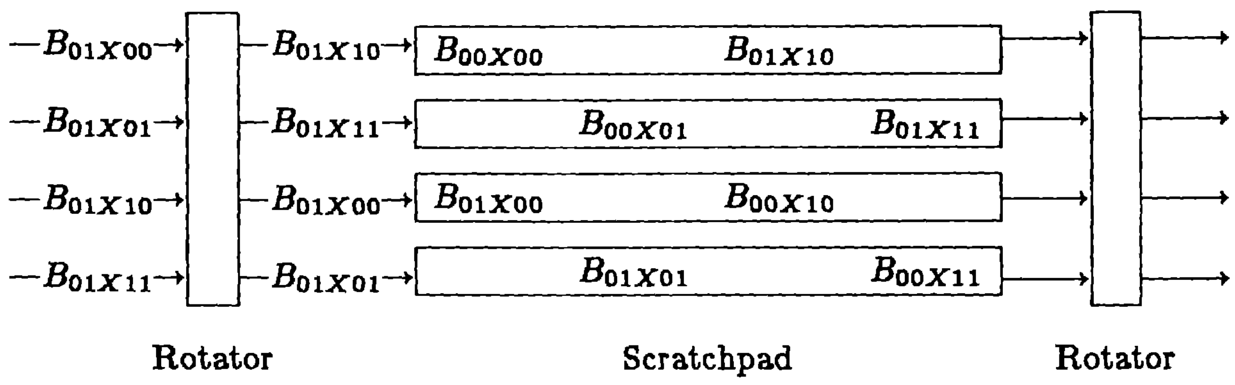


**Figure 7.11:** Reading The Subvector, Step 2

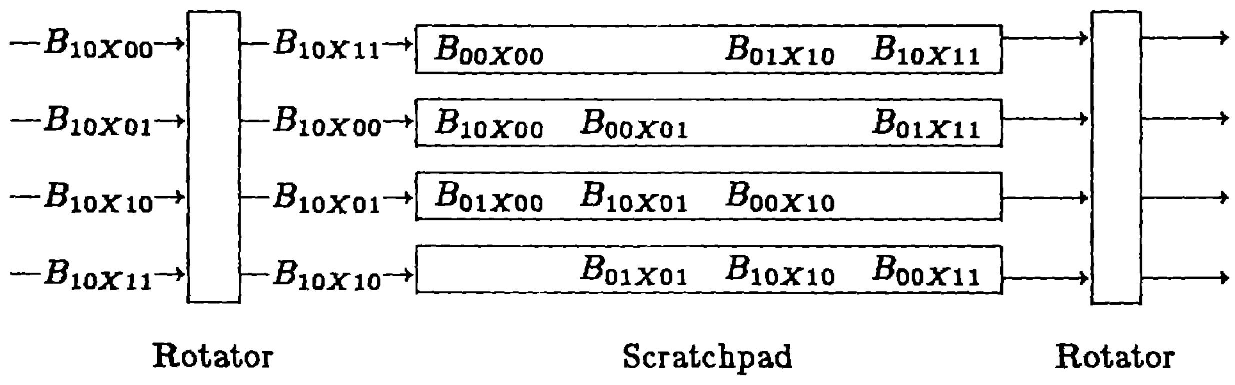


**Figure 7.12:** Reading The Subvector, Step 3

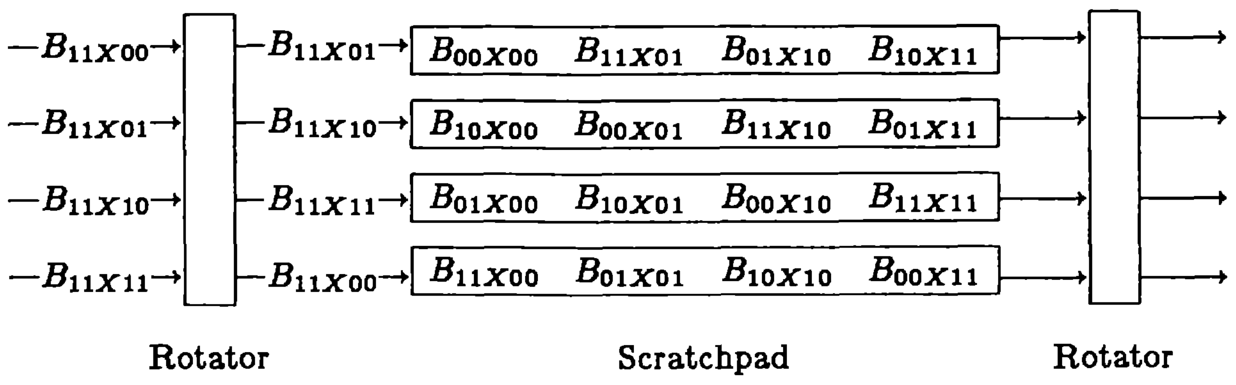

$B_{11}x_{00}$
$B_{11}x_{01}$
$B_{11}x_{10}$
$B_{11}x_{11}$
$B_{11}x_{01}$
$B_{11}x_{10}$
$B_{11}x_{11}$
$B_{11}x_{00}$
$B_{00}x_{00}$ $B_{11}x_{01}$ $B_{01}x_{10}$ $B_{10}x_{11}$
$B_{10}x_{00}$ $B_{00}x_{01}$ $B_{11}x_{10}$ $B_{01}x_{11}$
$B_{01}x_{00}$ $B_{10}x_{01}$ $B_{00}x_{10}$ $B_{11}x_{11}$
$B_{11}x_{00}$ $B_{01}x_{01}$ $B_{10}x_{10}$ $B_{00}x_{11}$
Rotator
Scratchpad
Rotator

**Figure 7.13:** Reading The Subvector, Step 4

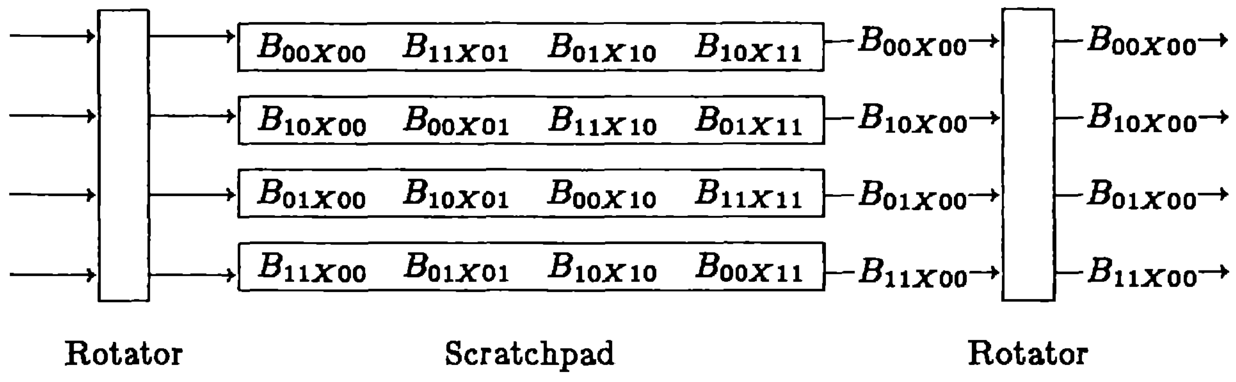

$B_{00}x_{00}$ $B_{11}x_{01}$ $B_{01}x_{10}$ $B_{10}x_{11}$
$B_{10}x_{00}$ $B_{00}x_{01}$ $B_{11}x_{10}$ $B_{01}x_{11}$
$B_{01}x_{00}$ $B_{10}x_{01}$ $B_{00}x_{10}$ $B_{11}x_{11}$
$B_{11}x_{00}$ $B_{01}x_{01}$ $B_{10}x_{10}$ $B_{00}x_{11}$
$B_{00}x_{00}$
$B_{10}x_{00}$
$B_{01}x_{00}$
$B_{11}x_{00}$
$B_{00}x_{00}$
$B_{10}x_{00}$
$B_{01}x_{00}$
$B_{11}x_{00}$
Rotator
Scratchpad
Rotator

**Figure 7.14:** Writing The Subvector, Step 1

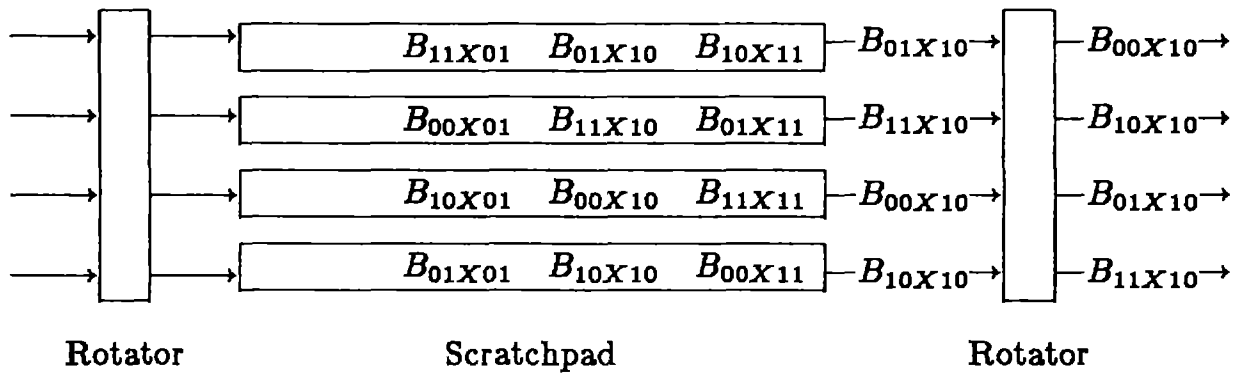

$B_{11}x_{01}$ $B_{01}x_{10}$ $B_{10}x_{11}$
$B_{00}x_{01}$ $B_{11}x_{10}$ $B_{01}x_{11}$
$B_{10}x_{01}$ $B_{00}x_{10}$ $B_{11}x_{11}$
$B_{01}x_{01}$ $B_{10}x_{10}$ $B_{00}x_{11}$
$B_{01}x_{10}$
$B_{11}x_{10}$
$B_{00}x_{10}$
$B_{10}x_{10}$
$B_{00}x_{10}$
$B_{10}x_{10}$
$B_{01}x_{10}$
$B_{11}x_{10}$
Rotator
Scratchpad
Rotator

**Figure 7.15:** Writing The Subvector, Step 2

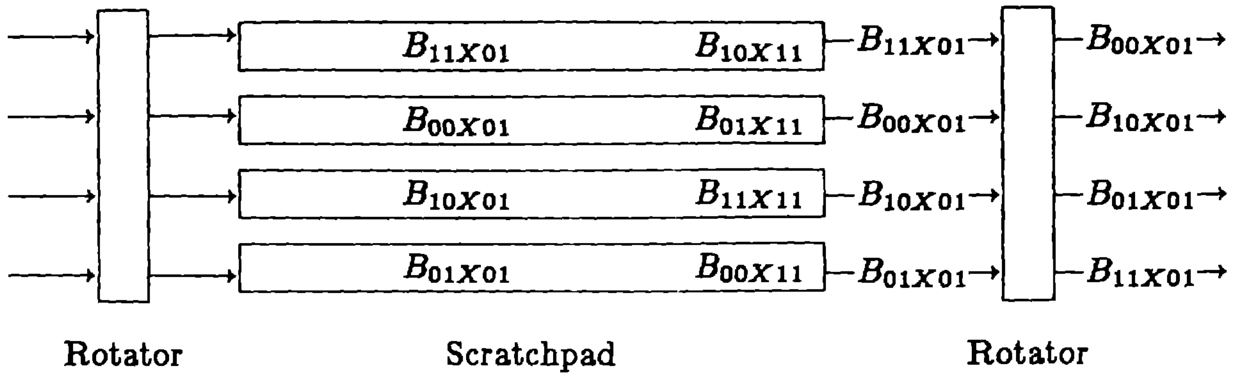

$B_{11}x_{01}$ $B_{10}x_{11}$
$B_{00}x_{01}$ $B_{01}x_{11}$
$B_{10}x_{01}$ $B_{11}x_{11}$
$B_{01}x_{01}$ $B_{00}x_{11}$
$B_{11}x_{01}$ $B_{00}x_{01}$ $B_{10}x_{01}$ $B_{01}x_{01}$
$B_{00}x_{01}$ $B_{10}x_{01}$ $B_{01}x_{01}$ $B_{11}x_{01}$
Rotator
Scratchpad
Rotator

**Figure 7.16:** Writing The Subvector, Step 3

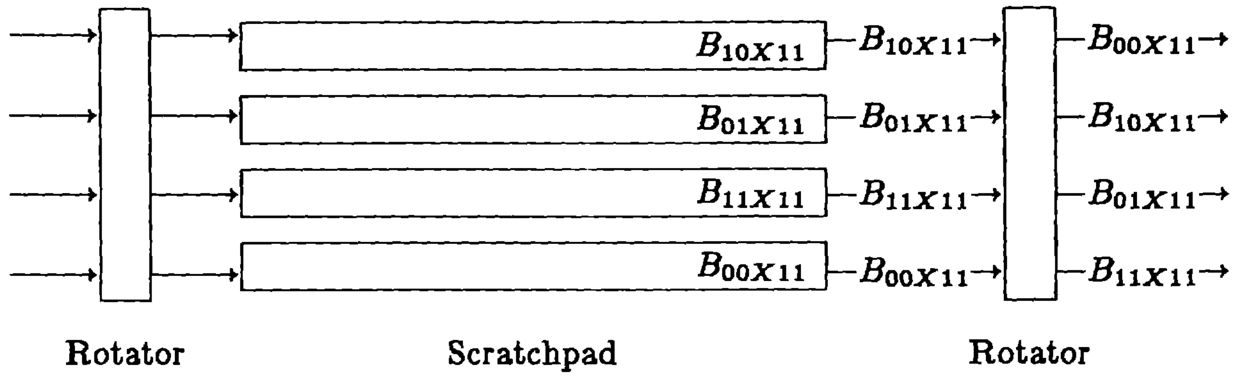

$B_{10}x_{11}$
$B_{01}x_{11}$
$B_{11}x_{11}$
$B_{00}x_{11}$
$B_{10}x_{11}$ $B_{01}x_{11}$ $B_{11}x_{11}$ $B_{00}x_{11}$
$B_{00}x_{11}$ $B_{10}x_{11}$ $B_{01}x_{11}$ $B_{11}x_{11}$
Rotator
Scratchpad
Rotator

**Figure 7.17:** Writing The Subvector, Step 4

## 7.2 Use Of The Technique.

Unless the matrices of interest are square, it is difficult to perform parallel transpositions in-place. Many applications algorithms only need to be able to access both rows and columns of the matrix in parallel, rather than requiring the explicit transposition to actually exist. The technique described in this dissertation can be used to provide parallel access to the columns of a row-stored matrix without actually forming its transpose; however, while supplying the data from one column, it must also provide the corresponding entries of the $N - 1$ other columns. By using the data of these "extra" columns immediately, rather than throwing them away, effectively parallel access to both rows and columns can be attained.

The ADI [Young73] algorithm offers a good example of the use of the transpose hardware. Without going into excessive detail, an ADI sweep consists of two passes over two-dimensional matrices. The first pass is a recurrence in the row direction, i.e., of the form

$$X[r,c] = f(X[r-1,c], Y[r-1,c]),$$

while the second is a recurrence in the column direction,

$$X[r,c] := f(X[r,c-1], Y[r,c-1]).$$

On parallel machines (either vector or array), it is advantageous to be able to access slices in the non-recurrence direction (i.e., columns for the first pass and rows for the second), since these operations on these slices are independent. However, with normal matrix representation and interleaved memory, either the column accesses or the row accesses cannot be made in parallel; the total ADI processing time is thereby substantially increased. With the hardware described above, one can approach the problem in one of two ways: either perform a full transposition after each pass, or else just perform local transpositions, as will be detailed below. The matrices used in ADI are frequently square, and in this case the full transpositions can be done in-place; however, this involves extra data movement and, in cached memory systems, inefficient use of the cache. The inefficiency of doing a full transpose depends upon the ratio of the arithmetic pipeline rate to the rate of the cache memory; for the S-1 Mark IIA, in which the ratio is unity, performing a full transpose costs a factor of two more than if the transpose and computation can be done together "on the fly".

Let us assume that the data is stored in row-major form (i.e., with the rows in sequential locations). The column recurrence is straightforward to implement (since the parallel slices are in the row direction). To implement the row recurrence pass, we need to impose conditions on the latency times of the execution unit. Let $T_l$ be the latency of computing $f$. (Here the latency is the total elapsed time from when the computation of $f$ begins to when the result of $f$ is available to begin the next computation; note that this may or may not include the full memory access and data alignment time, depending upon when the system has any sort of "short-stop" or "wrap-around" data paths.) Let $T_p$ be the the rate of computation of $f$ for separate data values (i.e., the pipeline time for vector machines or the execution time divided by the number of execution units for an array machine). With the assumption that $N*T_p \geq T_l$, we can implement the row recurrence pass of ADI by reading $N$ sequential $N$-word column slices (in $N$ memory cycles), then applying $f$ (in parallel) to the current $N$-word result slice and the first column slice (since there are $N$ different recurrences), and then to the result of the application and the second column slice, etc. Since we have assumed that $N*T_p \geq T_l$, we know that we will not have to wait for any previous result slice before beginning the next step of the recurrence.

## 7.3 Use Of The Skewing Memory As A Buffer.

The hardware configuration just described for transposing is ideally suited for buffering between the operand fetch unit of a CPU and its arithmetic unit. The memory used to store the skewed submatrices may also be used as a queue to allow the operand fetch unit to continue operations even though the arithmetic unit may be busy doing a lengthy computation, or to allow the arithmetic unit to catch up after such a period while the operand fetch unit is performing overhead functions unique to it (handling cache misses, complicated addressing modes, waiting for pipeline interlocks, etc.). This has been done on the Mark IIA in the Operand Queue (see Figure 3.1).

# 8. Pipelining Quicksort

The straightforward implementation of sorting algorithms on pipelined machines usually leads to very inefficient use of the hardware. For instance, the usual algorithm for the inner loop of Quicksort [Sedgewick75, Knuth73] is

```
    (* begin partitioning inner loop *)
    while true do
        begin
(*1*)   do h ← h-1 until A[h] < P;
(*2*)   do l ← l+1 until A[l] ≥ P;
        exitif h≤l;
        A[l] ↔ A[h]
        end;
```

The two loops marked with (*1*) and (*2*) would be compiled for the S-1 architecture into

```
L1:     SUB H=H,1
        SKP.GEQ A(H),P,L1

L2:     ADD L=L,1
        SKP.LSS A(L),P,L2
```

Now assuming that the original data of the array A were randomly ordered, the probability that either of the jumps will branch in the above two loops is one-half. As a class, pipelined machines take several different approaches to such unpredictable loops [Rau77, Kogge81a, Holgate80]. Most such approaches fit into one of three categories.

The first (and simplest) strategy is to simply stop further execution until the correct branch path has been determined. (Cray-1, CDC-7600)

The second strategy is to attempt to predict which one of the two paths will eventually be taken, and then to continue down that path as if the branch were unconditional. At some point in the pipeline, the correct path of the branch will be determined and then checked against the path previously chosen. If the earlier choice between the two paths was incorrect, the pipeline is backed up (i.e., the instructions executed down the wrong path are undone) and proceeding down the other, now-guaranteed-correct path is commenced. (Manchester MU-5 [Morris79], S-1 Mark I and Mark IIA, IBM 3033)

The third general strategy involves attempting to follow both paths. This usually involves fetching instructions from both paths until the correct course of the branch can be determined. In actual implementions, the difference between the second and third alternative often blurs, since an attempt may be made to fetch down both paths, but primarily from one.

Clearly the first works best with a very short pipeline, while the second works well as long as the prediction mechanism usually succeeds ([Smith81]). Although the third approach would seem to solve the problem, it works poorly if one or both of the paths being followed contain additional conditional branches. In order to be able to eventually have followed the correct path, all paths must be evaluated. The process which does this can easily get out of hand, since for any reasonable hardware implementation only a limited number of paths (usually one or two) can be followed concurrently.

Unfortunately, none of the above strategies work well for the Quicksort inner loop, since the branches are inherently unpredictable and are quite closely spaced. (For instance, see p. 229 of [Morris79] for a timing of Quicksort on the MU-5.) Furthermore, many pipelined machines gain much of their relatively high performance by fetching vectors (ordered streams of data) from memory, which feature is not at all useful with the straightforward implemention of the Quicksort inner loop.

Methods of using vector instructions for sorting have been proposed, and at least one has been implemented on the Cray-1 [Sedgewick78 and Kulsrud78], resulting in the fastest sorting program known to the author. They all are somewhat complex and none allow the inner loop to execute at the memory bandwidth-limited rate.

We now demonstrate a parallel pipelined structure which quite effectively implements the Quicksort inner loop; moreover, minimal hardware is required to implement this structure.

The read address generation hardware in Figure 8.1 has two pointers, $L_r$ and $H_r$, corresponding to l and h of the Quicksort program. These pointers initially point to the beginning and end of the array. The read-address hardware initially reads $T + S$ words from either end of the array, using either the $L_r$ or $H_r$ pointer (it doesn't matter which), modifying the pointers appropriately. After the $T + S$ cycles, the read-address hardware switches to a mode wherein it uses the $L_r$ or $H_r$ pointer, depending upon a single-bit value taken from a queue between the data comparator and the read-address hardware. $S + 1$ or more cycles after the comparison takes place (and thus after the reading operation which is dependent upon the result of the compare occurs), the data are written into the location pointed to by $L_w$ or $H_w$ of the write-address hardware (which is also modified appropriately). Since the write to a given location occurs after the fetch from that location, we have ensured that $L_w < L_r$ and $H_r < H_w$, thus assuring we don't write over input data which hasn't been processed. The algorithm terminates after $L_r \geq H_r$ (or, equivalently, the same number of read cycles as the size of the array have occurred).

The algorithm just specified (Pipelined Quicksort 1) works well for pipelined systems in which memory writes and reads have the same cost. However, consider a system in which a memory write involves both a read and a write cycle. This could occur, for instance, on a system with a cache, where the cache is implemented such that a write requires a quasi-read cycle to ensure that the location to be written is indeed in the cache. On such a system, it is very advantageous to be able to write back into the same locations as were originally read (and in the same order).

We now describe a modified algorithm, Pipelined Quicksort 2, for such a cached computer system (see Figure 8.2). In this algorithm, there is

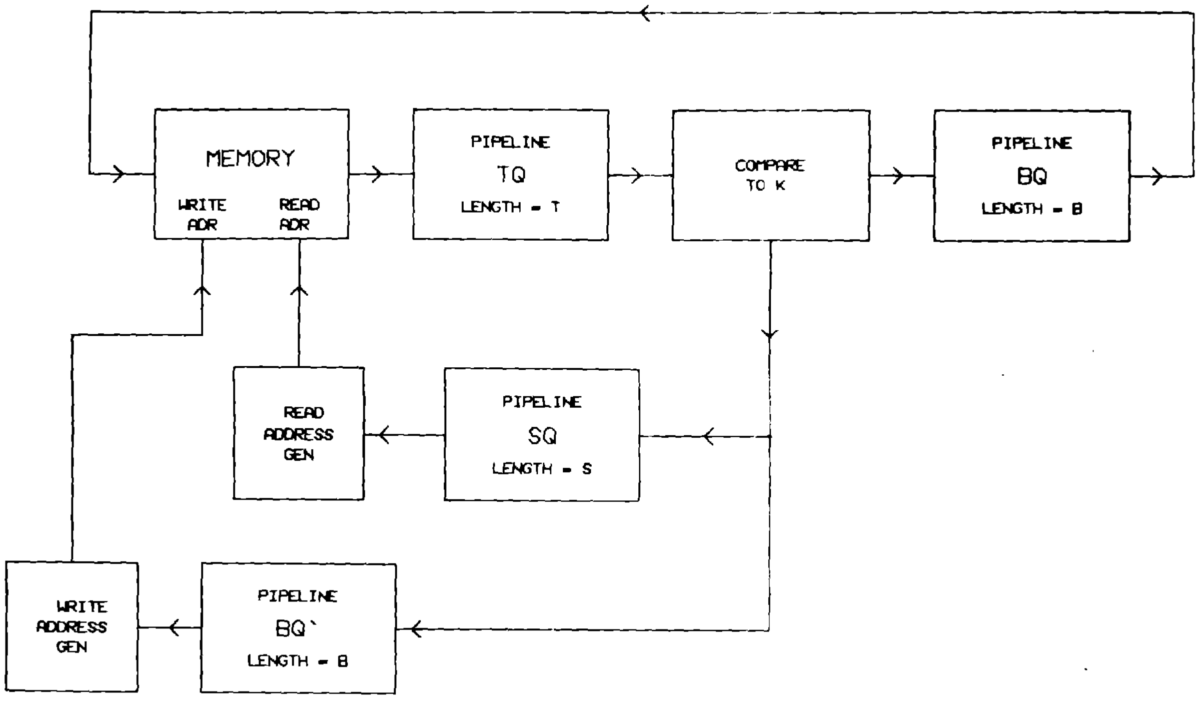


**Figure 8.1:** Pipelined Quicksort 1

only one address generator, the read-address hardware of Figure 8.2, and the write address is simply a queue of delayed addresses from the read-address generator. The pointers in the address hardware are initialized as before. The first $T + S$ cycles again are reads from, say, the low end of the array, with no associated writes. As before, the read-address hardware also proceeds into the same mode of looking at bits coming from the comparator and reading from the array, using pointer $H_r$ if the bit is one and from $L_r$ if zero. Now, however, writes are set up (i.e., the addresses are placed in DQ) for all reads.

Since each write corresponds to the result of a compare, we are assured of having data (in the correct order) to be written, simply by using

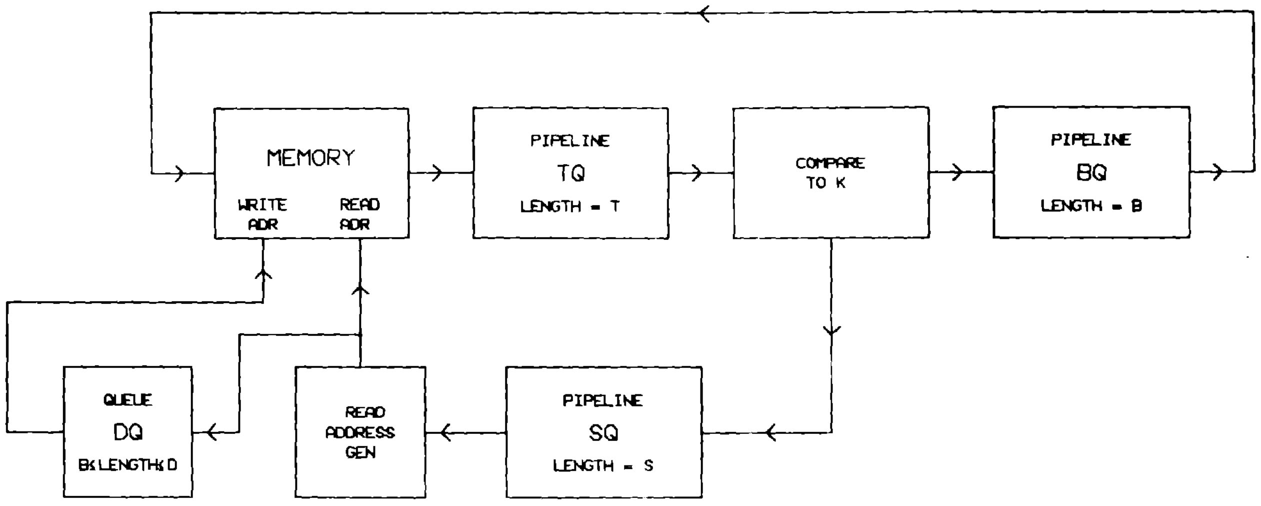


**Figure 8.2:** Pipelined Quicksort 2

the original data stream. The only requirement is that $b \geq s$, which is necessary to ensure that we can hold the data long enough to have found locations in which to put it.

After $N$ cycles, we have read the entire array (since we are reading one word every cycle). We have only scheduled $N - T - S$ writes and so there are $T+S$ "extra" operands in the pipeline with (as yet) no place to be written. To find somewhere to put these data, we now put the read-address hardware into a new mode. The $L_r$ pointer is first reset to the origin of the array (without modifying the $H_r$ pointer). We now continue to examine bits from the comparator, using the $L_r$ or $H_r$ pointer, depending upon the value of the bit. However, the action we take is slightly different than before. When using the $L_r$ pointer, we are only reading in order to set up a write; when using the $H_r$ pointer, we always set up the write and only do the read if $H_r \geq T + S$. This means that we are only willing to read data that have already been written on the first pass. The read-address hardware then proceeds in this manner until TQ is empty, at which time it is free to proceed to other work.

The following is a trace of this algorithm for ten points, with s=2, t=4, b=3, and a partitioning key of 14. The |'s mark the location of $L_r$ and $H_r$, and a circled value means that a write is currently scheduled to that location. TQ, BQ, and SQ are the contents of the corresponding pipelines in Figure 8.2.

```
Cycle           Array                          Contents of Pipelines

0      |20  73 61  32 11  19   7 71 64  5|      [tq]
                                                [bq]
                                                [sq]
1      |20  73 61  32 11  19   7 71 64  5|      [tq 20]
                                                [bq]
                                                [sq]
2       20| 73 61  32 11  19   7 71 64  5|      [tq 20 73]
                                                [bq]
                                                [sq]
3       20  73|61  32 11  19   7 71 64  5|      [tq 20 73 61]
                                                [bq]
                                                [sq]
4       20  73 61 |32 11  19   7 71 64  5|      [tq 20 73 61 32]
                                                [bq]
                                                [sq]
5       20  73 61  32|11  19   7 71 64  5|      [tq 73 61 32 11]
                                                [bq 20]
                                                [sq 1]
6       20  73 61  32 11| 19   7 71 64  5|      [tq 61 32 11 19]
                                                [bq 20 73]
```

```
                                                        [sq 1 1]
7      20  73 61  32 11  19|  7 71 64|5                [tq 32 11 19 5]
                                                        [bq 20 73 61]
                                                        [sq 1 1]
8      20  73 61  32 11  19|  7 71|64 20               [tq 11 19 5 64]
                                                        [bq 73 61 32]
                                                        [sq 1 1]
9      20  73 61  32 11  19|  7|71 73 20               [tq 19 5 64 71]
                                                        [bq 61 32 11]
                                                        [sq 1 0]
10     20  73 61  32 11  19||7 61 73 20                [tq 5 64 71 7]
                                                        [bq 32 11 19]
                                                        [sq 0 1]
11     20 |73  61  32 11  19 |32 61 73 20              [tq 64 71 7]
                                                        [bq 11 19 5]
                                                        [sq 1 0]
12     11| 73 61  32 11|  19  32 61 73 20              [tq 71 7]
                                                        [bq 19 5 64]
                                                        [sq 0 1]
13     11 73 |61  32 11| 19  32 61 73 20               [tq 7]
                                                        [bq 5 64 71]
                                                        [sq 1 1]
14     11   5|61  32|11 19  32 61 73 20                [tq]
                                                        [bq 64 71 7]
                                                        [sq 1 0]
15     11   5|61 |32 64  19  32 61 73 20               [tq]
                                                        [bq 71 7]
                                                        [sq 0]
16     11   5 61||71 64  19  32 61 73 20               [tq]
                                                        [bq 7]
                                                        [sq]
16     11   5  7||71 64  19  32 61 73 20               [tq]
                                                        [bq]
                                                        [sq]
```

A similar algorithm works when the memory can read and write $P$ sequential words per cycle. In this case, the comparator must compare multiple words per cycle (either in parallel or in a pipeline) and send a bit back to the read-address hardware whenever it gets at least $P$ words for the high or low partition. This will require one more cycle than in the previous algorithm to get started, but we are guaranteed to have such a block each cycle thereafter, since if we have $2P$ words which are divided into two sets (those destined for the high partition and those destined for the low partition), at least one of the sets must contain $P$ or more elements. We now assemble a $P$-word block (leaving the rest of the words for future cycles) and write it back to memory at the previously scheduled address.

Thus we can perform a Quicksort as fast as the interleaved memory can cycle, leading to quite high processing rates.

# 9. Pipelined Computation On Cylinders

As discussed above, the total execution latency of the arithmetic operations is a very important consideration in determining the performance of a pipelined system. We will now consider a three-dimensional positioning of pipelined arithmetic hardware which helps keep this latency as small as possible, as well as a method of scheduling the holding latches of the pipeline which minimizes the timing criticality of certain global "pipe-stopping" signals.

This dissertation is not the appropriate place for a detailed discussion of pipelined digital system techniques. The reader is referred to [Kogge81] for an excellent description of such systems. [Cotten65] and [Cotten69] are good short papers on the timing of this type of digital circuitry.

## 9.1 The Topology of Pipelined Systems.

First, we observe that nearly all current hardware implementation technologies constrain one to build with two-dimensional structures (wire-wrap boards, PC cards, VLSI chips, etc.) interconnected at their edges.

We need to now consider a realizable topology suitable for constructing our pipelined arithmetic hardware.

Consider the hardware necessary to accomplish multiplication. Since we are going to pipeline this hardware, its design will resemble that of Figure 9.1. Since the flow of data is strictly forward, a natural choice for physically positioning the hardware necessary to implement this design is to follow the same strategy: build each combinatorial network and connect them together such that the data flows from (say) left-to-right. Thus we are computing on a rectangular array of logic elements with the inputs arriving on the array's left edge, with the computation flowing generally towards the right, and with the outputs leaving from the right edge (Figure 9.2).

The total execution latency, however, includes the time it takes for the signal to propagate from the result (arising at the right edge) to the input (at the left edge). The right edge is the first location from which a wrap-around can commence: it is where a result is first available to wrap from. An obvious way to minimize the wrap-around delay is to deform the rectangle into a cylinder, with its right and left edges joined together as a "seam".

None of the considerations leading to the cylinder construction involved anything unique to the structure of a multiplier; thus, one is led to the same construction for all of the computing units of the pipelined arithmetic hardware. However, the total latency time must include the worst time from any output to any input. Thus, we are led to take all of the cylinders and glue all of their seams together (or at least as closely as

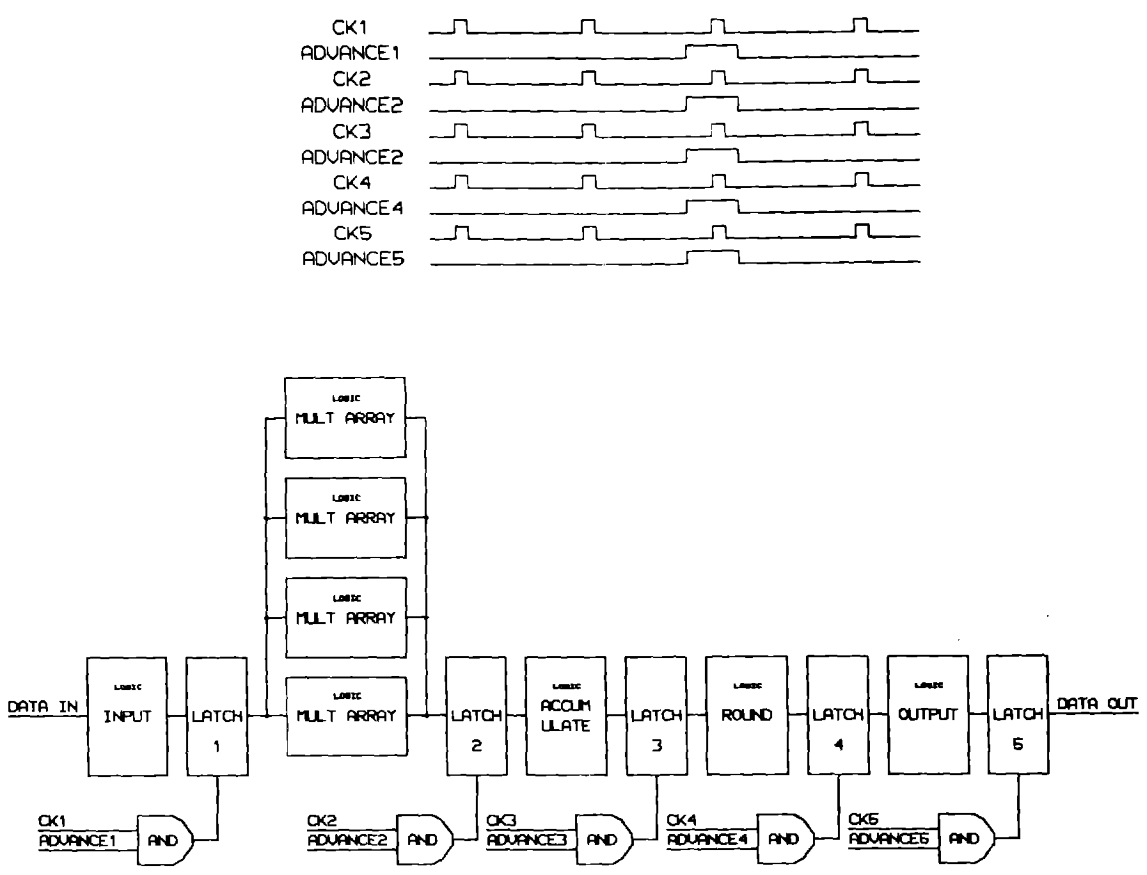


**Figure 9.1:** A Simple Pipelined Arithmetic Structure: A Multiplier

physically possible). As a specific example, Figure 9.3 shows an idealized picture of the topology of the S-1 Mark IIA arithmetic unit.

Consider now the *advance* signals in Figure 9.1. These signals tell the computation at each stage of the arithmetic pipeline whether or not it is allowed to proceed. Note that *advance* signals are pipe-stopping signals — the absence of an *advance* signal means to stop that segment of the pipeline. (The values of *advance* shown advance the pipeline once, i.e., a "single-step"

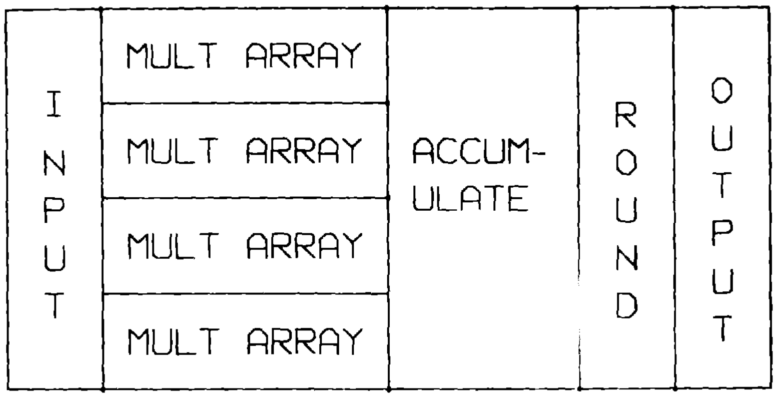


**Figure 9.2:** Idealized Layout Of Figure 9.1

is indicated.) Some machines (e.g., the Cray-1) don't have such signals; instead, by the time an operation has made it to the computation section ("issued" in Cray parlance), it has been checked and it is known that the operation can flow unimpeded until it has finished (i.e., that there are no possible future hazards for this operation). Such guarantees are difficult to implement on more complicated machines, such as those with a memory hierarchy (possibly including caches) or those which implement complex microcoded operations.

The several conditions (e.g. floating-point overflow) whose occurrence can require halting of the arithmetic pipeline have natures such that they usually can be recognized just before stopping of the pipeline is required by their occurrence. Assuming that we have some flexibility, we can place the logic necessary to determine these stop conditions and to generate the *advance* signals from the various stop conditions close to the seams of the

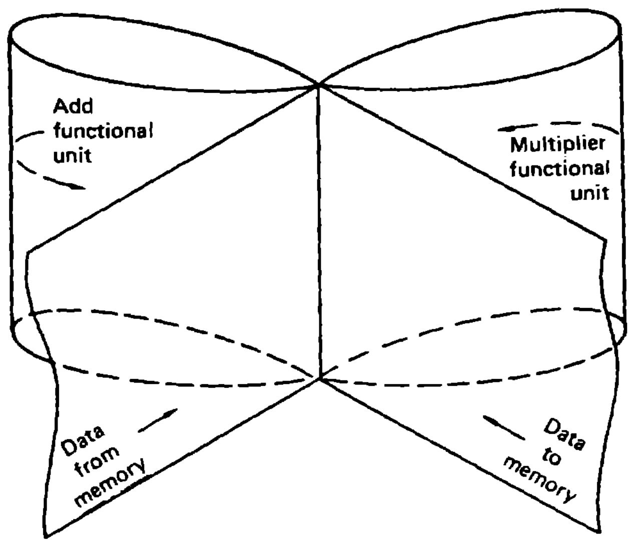


**Figure 9.3:** Idealized Topology Of S-1 Mark IIA Arithmetic Unit

cylinders described above. This works well for the first and last pipeline registers of Figure 9.1, but what about the stages in the interior (away from the edges) of the cylinder? They are a significant distance from the edges (where distance is measured in propagation delay of the signals which carry the information of interest), and thus getting the *advance* signals to the pipeline registers in the interior of the cylinders in time to correctly control the pipeline could easily become a limiting consideration in increasing the speed of a machine. (We have assumed that there is no substantially shorter path for the *advance* signals than to follow the surface of the cylinder. This is nearly always the case.) We now show a method of modifying the clock timing of the interior registers such that the interior versions of the advance

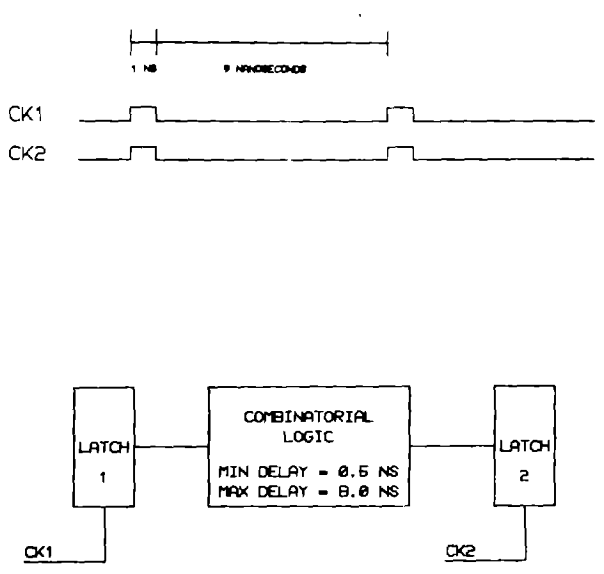


**Figure 9.4:** Example Of Failure Of Simple Pipeline Structure

signals can arrive later than would otherwise be necessary, and hence the additional delay of traversing the surfaces of the cylinders is much less of a problem.

A simple approach to pipelining as diagrammed in Figure 9.4 is seldom sufficient in practice, especially when the pipeline stages are implemented with latches for attainment of highest speed. The problem is that both real components and real wires have skew; their maximum propagation delay is not equal to their minimum delay. Since a latch must be held open for some period of time to allow the signal value input to it to stabilize, there exists the very real problem of having sections of logic where the

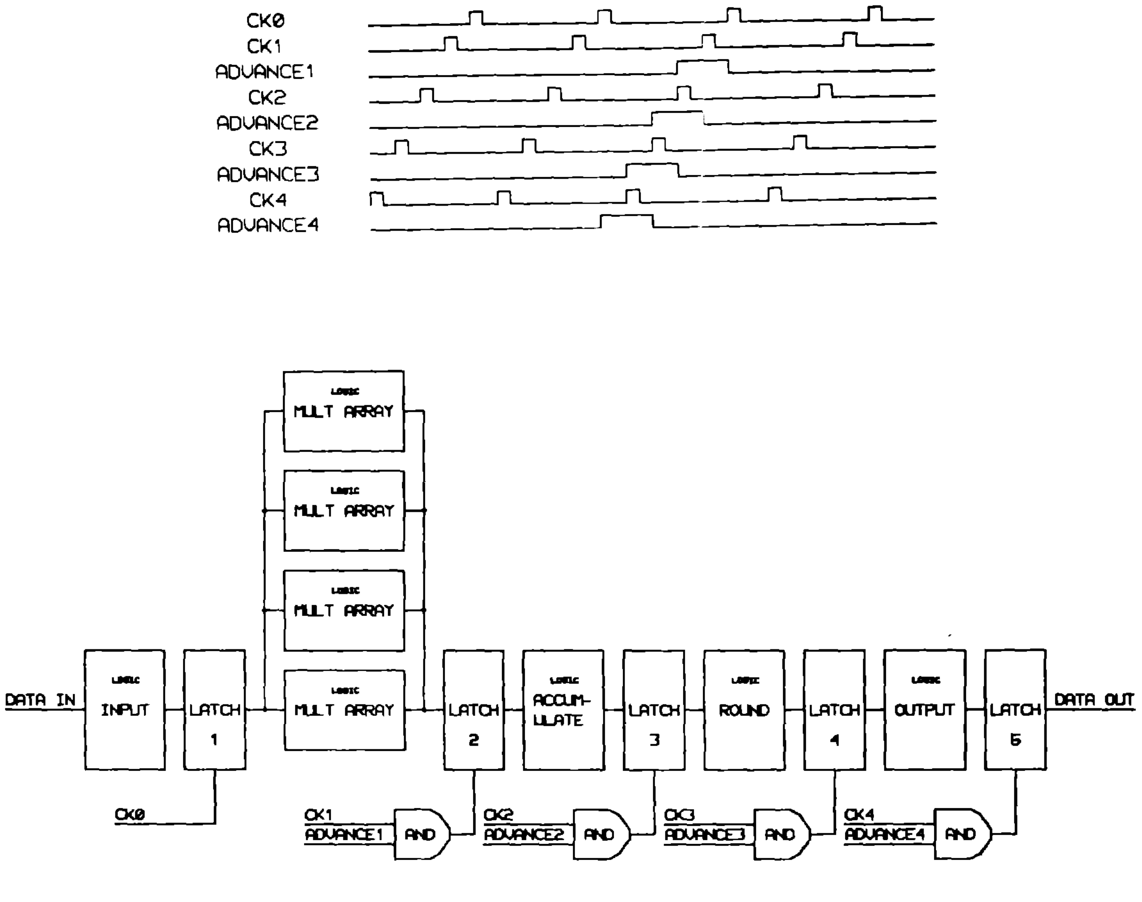


**Figure 9.5:** More Realistic Pipelined Structure

minimum propagation time decremented by the time the latch must be held open is negative, which will at least sometimes lead to the pipeline's output register being loaded with incorrect data! (See Figure 9.4 for an example, where the latches are "idealized", i.e., the setup and hold times are zero, as are the propagation delays from clock to output and from input to output. The delay of real latches can be modelled by adding skew to the clocks and adjusting the minimum and maximum delays of the combinatorial logic network.)

For these reasons, a more realistic arithmetic pipeline might look like that sketched in Figure 9.5 (and where, again, a "single-step" is shown). Note that there can be more registers than pipeline stages, and that the registers are not necessarily clocked with the same clock phase. Furthermore, only some of the registers have *advance* signals.

What has been done is to designate certain of the registers as pipeline holding registers. If the pipeline is stopped, then these registers hold the data needed to resume the computation. Clearly, for an $s$-stage pipeline, there need to be exactly $s$ holding registers. The extra registers are there to provide time-wise stability for the flow of data, i.e., to remove the effects of skew from the combinatorial networks at each pipeline stage.

The choice of which pipeline registers to use as holding registers is weakly constrained. Note, however, that we have located the holding registers as "late" as possible. This is important as it means that the corresponding *advance* signal can be late also. The last stage must be exactly on a cycle boundary, since all of the arithmetic functional units must produce their outputs at the same time and at multiples of cycles. However, the last stage is the closest stage to the *advance* signal generation logic, while the stages inside the cylinder (which are further from this logic) are delayed, and thus need later versions of *advance*. This is exactly the desired result: we have significantly lessened the time-criticality of *advance*.

To aid the designer in implementing such a circuit, it is useful to have a notation which describes signals in such a way that illegal combinatorial networks may be readily detected. For example, Figure 9.6 illustrates a

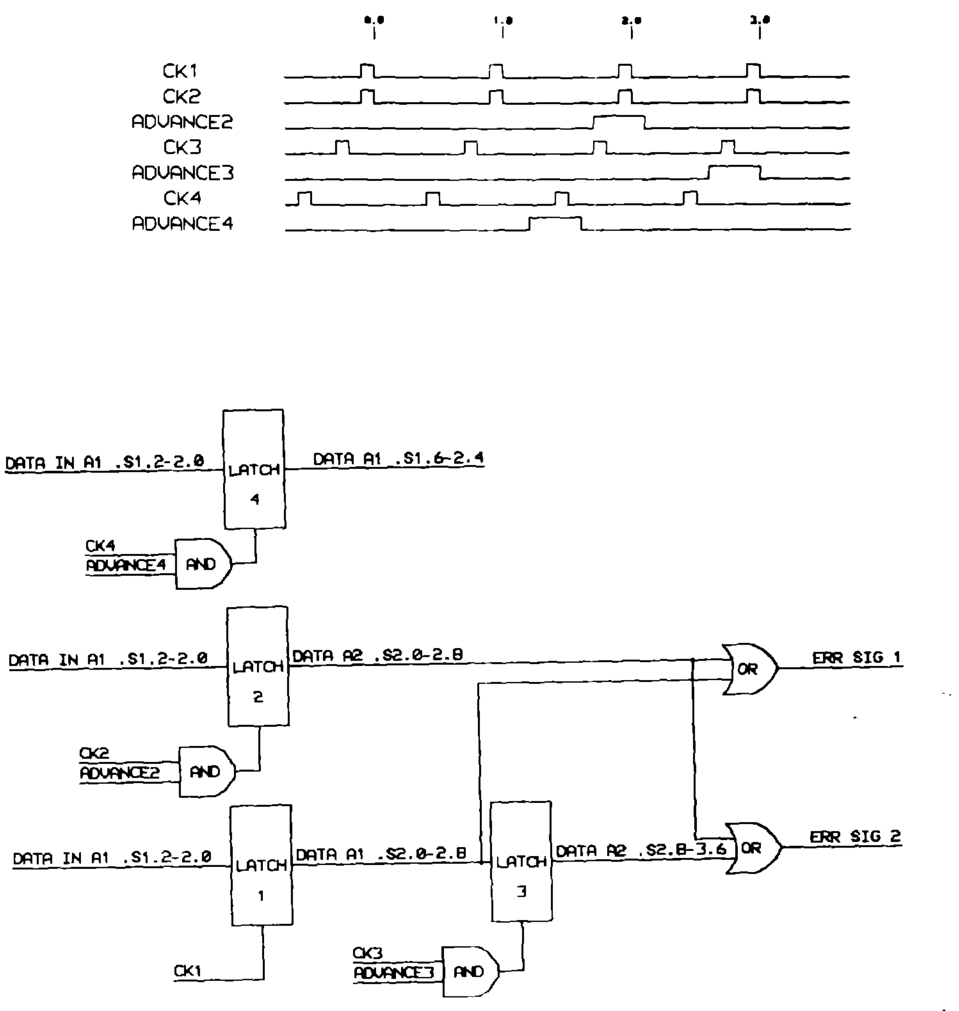


**Figure 9.6:** Example Of Incorrect Combination Of Signals

circuit which combines signals from different pipeline stages (ERR SIG 1) and which combines signals from the same pipeline stage but with inconsistent time delays from the pipeline's origin (ERR SIG 2).

The naming convention developed for use in the S-1 Mark IIA

ABOX design takes this design methodology into account, and consists of three components: a signal name, a pipeline stage and a physical time specification. For example, in the signal **DATA A2 .S2.0-2.8** the signal name is **DATA**, the signal is in the second stage of the **A** pipeline and the signal is stable (at least) between 2.0 cycles and 2.8 cycles after the origin of the pipeline. Figure 9.6 shows various versions of the signal **DATA** and how they would be generated. Notice that the signal **DATA A2 .S1.6-2.4** is a very early version of a second pipeline stage signal which would require a comparably early version of *advance*, and thus would probably not be feasible to generate.

This notation (without the pipeline stage indication) and a system for automatically verifying the timing of designs is described in [McWilliams79]; the pipeline stage legality check has been done manually in the S-1 Mark IIA design work, but clearly could be automated.

# 10. Execution Out-of-Order Without Imprecise Interrupts

Many high performance machines have allowed the execution of instruction streams "out of order" to increase performance. Unfortunately this leads to a class of phenomena known as "imprecise interrupts" [Anderson67, Kogge81a], which make it difficult to deal with exception conditions that arise as a result of operations within the pipeline. We discuss here an approach to "out of order" execution which obviates the imprecise interrupt problem.

We first define what is meant by "out-of-order" and "imprecise." The model of the instruction stream for a conventional architecture is sequential, i.e., instructions can be considered to be executed serially. Therefore, let us call the instruction sequence $I_n$ for $n > 0$. A pipelined machine, of course, attempts to perform the instructions in parallel while producing the same result as if the serial nature of the stream had been preserved. If every pipeline stage processes instructions in strict order (i.e., $I_{n+m}$ is never performed before $I_n$ for $m > 0$), then the execution of each stage is serial and hence the operation of the pipeline as a whole may be considered to be "in order".

Conversely, if *any* stage in the pipeline admits the possibility of $I_{n+m}$ being performed before $I_n$, then we say that the execution of instructions by the pipeline is "out-of-order".

Note that there is nothing inherent in the definition of an "out-of-order" pipeline which prohibits such a pipeline from producing the same results as would strictly serial processing of the instruction stream, nor does the definition of the "in-order" imply the same detailed functioning as a strictly serial execution computer.

## 10.1 Imprecise Interrupts.

The choice of the $n + 1^{st}$ instruction, $I_{n+1}$, may depend upon $I_n$ in one of many ways. We will consider five.

$I_n$ may unconditionally specify $I_{n+1}$ (e.g., a non-branching instruction is normally followed by the next instruction in memory). This is an *unconditional branch*.

The choice of $I_{n+1}$ may be from a (usually small) set of possibilities specified by $I_n$, and the actual choice made as a function of the state of the machine (before execution of $I_n$). This is a *conditional branch*.

$I_{n+1}$ may not depend upon $I_n$ at all; rather it was selected as a result of some event external to the pipeline. Such a break in the flow is an *interrupt*. In general, the state of the machine after the execution of $I_n$ is saved so that at some later time the instruction sequence can be resumed as if the external event had not occurred.

$I_{n+1}$ may be chosen because the execution of $I_n$ left the machine in some "unacceptable" state (such as with a word in memory not correctly representing the true result of the operation which was to be performed, such as overflow from an addition). This is a *trap-after-execution*.

Similar to *trap-after-execution* is *trap-before-execution*, where $I_{n+1}$ is chosen because the instruction that would normally have been after $I_n$, say $I'_{n+1}$, would have left the machine in an "unacceptable" state. This is essentially equivalent to tentatively performing $I'_{n+1}$, checking the machine state that results and then undoing $I'_{n+1}$ if the resulting machine state was "unacceptable". Note that *trap-before-execution* is nearly always more useful than *trap-after-execution*, since $I'_{n+1}$ may, if allowed to change the state of the machine, destroy information necessary to reconstruct the state before $I'_{n+1}$, thus making debugging or recovery more difficult.

Finally, we say that a pipeline exhibits an *imprecise interrupt* at $I_n$ for the sequence $I_j$, $(1 \leq j \leq n)$, if no sequence of instructions after $I_n$ can reconstruct the machine state that would have existed immediately after $I_n$ in the serial execution of $I_j$. Intuitively, the pipelined computer has executed $I'_j$ for $j > n$ which should not have been executed, due to a trap or interrupt occurring at the time of $I_n$.

General purpose computers do not usually exhibit imprecise interrupts, except within instruction sequences containing traps. Allowing imprecise interrupts in the first three types of instruction sequencing dependencies listed above would be unacceptable, since all occur quite often, and producing different results from the serial model of execution would not be of great utility. Many high performance machines have permitted imprecise interrupts in sequences containing traps, reasoning that a trap implies a fatal error such that the continued correct execution is not necessary (or, at least that the gains in performance of pipelines which exhibit such imprecise interrupts are more important than the correct execution of such "rare" cases). Machines exhibiting such imprecise interrupt behavior include the IBM 360/91, the TI ASC, the CDC Star 100 and the Cray-1.

The appropriateness of handling traps imprecisely is arguable. Taking this liberty in design eliminates (or at least makes very difficult) the possibility of smart trap software which attempts to continue when the error can be safely ignored or when the instruction sequence can be executed in some other manner which avoids trapping (e.g., in higher precision). It also means that the handling of rare but nonfatal errors must be performed by the hardware. Extending the number system implemented by the hardware with software which is only called into play when necessary is also difficult or impossible when imprecise interrupts are allowed. Clearly, IBM has felt that imprecise interrupts are too high a price to pay for increased pipeline speed, since the 360/91 was the only computer in the 360/370 line which ever exhibited it.

However, it is possible to have the performance advantages of out-of-order execution without paying the price of imprecise interrupts. In essence, we must ensure that the state of the machine at any point in time is equivalent to the state produced after some $I_n$ where the execution had proceeded serially. (Here equivalent means "exactly the same as" or "easily transformable to.") Clearly, the state at successive points in time must correspond to increasing (but not necessarily sequential) values of $n$.

To implement this, we allow temporary changes of state, i.e., state changes which allow successive instructions to proceed correctly, but which can easily be undone. Temporary state changes must eventually be made permanent, but only after it has been determined that the instruction causing the temporary change (and all preceeding instructions) will definitely be executed. Thus, if we determine that some instruction $I'_n$ was incorrectly chosen and that $I_n$ should have been the next instruction, we wait until all temporary changes made by instructions $I_k$ $(k < n)$ have been made permanent and remove all temporary changes made by instructions $I'_j$ $(j \geq n)$. We may then properly proceed with the execution of $I_n$.

## 10.2 Three Example Systems.

To better illustrate these ideas, we now give three example systems. The first is a simple in-order pipeline which doesn't exhibit imprecise interrupts, and the second and third have equal performance; the second exhibits imprecise interrupts, while the third doesn't. The extra hardware needed to implement this additional functionality of the third example is quite small, as will be shown.

### 10.2.1 In-Order Pipeline Without Imprecise Interrupts.

It is relatively straightforward to implement an in-order pipeline without imprecise interrupts. Let us call the first pipeline stage at which the next instruction to be executed is known (and not just guessed, as on a system with branch prediction) the *control point*; this is similar to the definition used in [Holgate80]. By constructing the pipeline such that no permanent state changes are made until after the control point, it is clear that imprecise interrupts can always be avoided. The Mark IIA is such a system, for instance.

### 10.2.2 Out-of-Order Pipeline With Imprecise Interrupts.

The IBM 360/91 is the classic example of a computer which can execute instructions of-of-order but which exhibits the phenomenon of imprecise interrupts. Other computers exhibiting this behavior include the CDC 6600, 7600 and the Cray-1. A block diagram of such systems might look like Figure 10.1.

### 10.2.3 Out-of-Order Pipeline Without Imprecise Interrupts.

Figure 10.2, although quite similar to Figure 10.1, does not exhibit imprecise interrupts. Instead, stores (and other state changes) of instructions are kept in temporary memories which allow the computation to proceed as in the pipeline of Figure 10.1. When the execution of instruction $I_n$ has completed, in the sense that the instruction which must follow $I_n$ has been determined (which, among other things, means that it has been determined whether or not $I_n$ causes a trap-after-execution or whether $I'_{n+1}$ will cause a trap-before-execution), we allow the permanent writes of all instructions $I_j$ with $j \leq n$ to proceed. At any point, if it is determined that the predicted sequence of instructions was incorrect, the contents of the temporary registers are destroyed and the correct sequence is initiated.

Figure 10.3 shows how the Mark IIA ABOX might be modified to implement the structure of Figure 10.2. A new instruction can be taken every cycle from the IBOX. If the functional units are busy or the data necessary to compute the new instructions are unavailable, then the instruction and its known data are saved. All results from both the adder and the multiplier are saved in all four temporary registers. In this way, the ordered nature of the permanent writes to be sent back to the IBOX can be ignored by the computations within the ABOX.

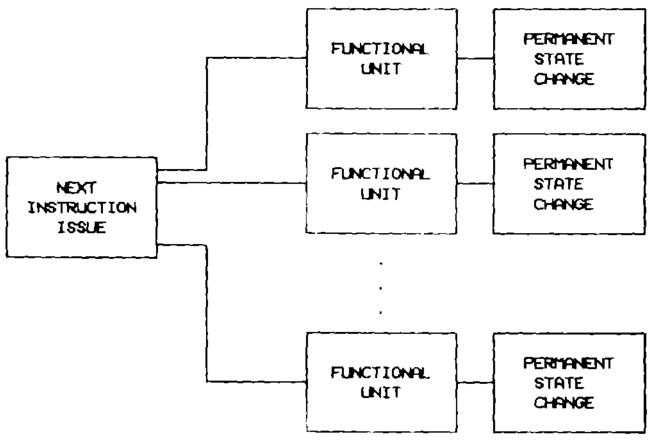


**Figure 10.1:** Out-of-Order Pipeline with Imprecise Interrupts

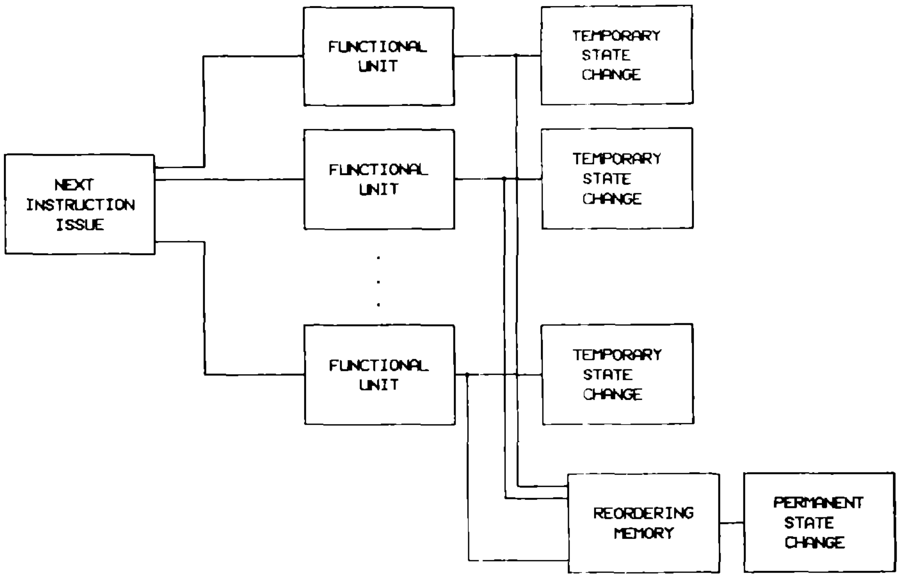


**Figure 10.2:** Out-of-Order Pipeline without Imprecise Interrupts

## 10.3 Conclusions.

We have shown a relatively straightforward method of removing imprecise interrupts from an out-of-order pipeline. This allows the construction of machines which have the highest performance available within the pipelined machine context, yet also feature the ease of debugging and recoverability of a precise interrupt machine.

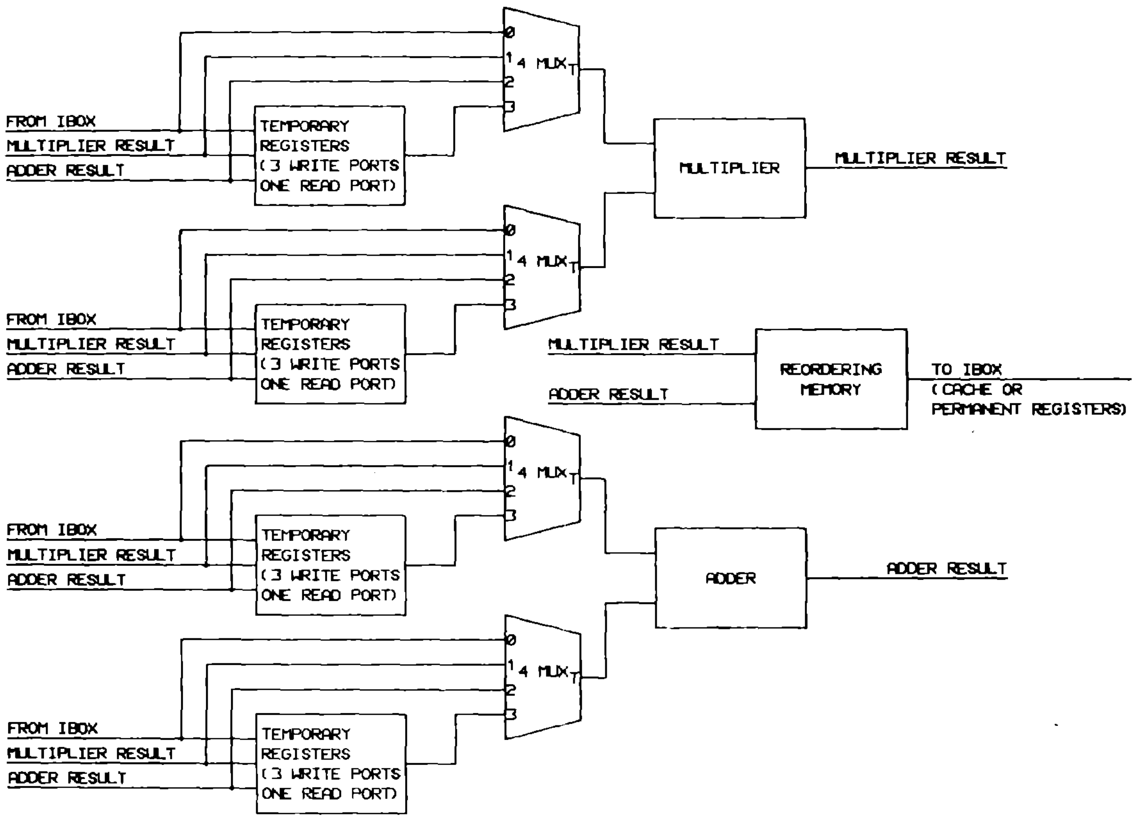


**Figure 10.3: Structure of an ABOX Permitting Out-of-Order Execution**

# 11. Contributions, Conclusions and Future Work

## 11.1 Summary.

A number of related techniques of importance to the design of high-performance parallel pipelined computer has been described in this dissertation, and their consequences for performance analyzed.

In addition to the new algorithms developed (recapitulated in the next section), two major new approaches to attainment of high computing throughput without sacrifice of data processing sophistication have been presented.

The first is the theoretical importance (and practical utility) of having separate data formats: the set of external or architectural formats actually seen by the programmer, and the set of internal formats actually used by the arithmetic hardware to implement the operations of the computer.

The second is a methodology for efficient implementation of an arithmetic pipeline–the concept of computing around cylinders. It is shown that this approach to hardware layout leads to relatively short interconnections between functional units, while simplifying the layout and timing of the logic used to implement them. A concise notation useful for describing the signals on such cylinders is also characterized.

## 11.2 Contributions Of The Present Work.

Several significant and interrelated contributions to the general problem of how to design and implement digital computing systems which feature very high throughputs across a wide spectrum of problems and applications have been made in the course of the research described in this dissertation.

The elementary function evaluation algorithm developed in the present work, though simple, appears to be novel, and obviates a major bias which previous generations of supercomputers have introduced into scientific computation, namely, that deriving from the very high cost of computing the standard elementary functions relative to the cost of addition or multiplication. The elementary functions of most importance in this regard are divide, square-root, exponential and logarithm. The great expense of evaluating such functions has occasioned much work in the development of algorithms which act to remove such functions from inner loops of scientific computations, often at a large cost in understandability, extensibility or accuracy of such programs. A computer in which evaluation of such functions entails costs nearly equal to that of addition or multiplication will surely be much easier to efficiently use in all such applications.

The floating-point addition algorithm developed in the present work results in a smaller latency for pipelined addition than previous approaches and is a significant step in maintaining the necessary balance between pipeline rate and latency so necessary for efficient operation of a general purpose arithmetic engine. The second aspect of the algorithm, the possibility of simultaneously generating sums and differences, significantly enhances the performance of processors using it to execute the FFT algorithm (and of the newer discrete Fourier transforms developed by [Winograd78]), which in turn is of fundamental importance in many applications.

The use of a small but versatile skewing memory within a processor's pipeline to implement transposition and bit-address-reversing is original to the present work. The use of skewed storage to efficiently access rows and columns of a matrix is an old idea, but the use of skewed storage for doing the bit-address-reverse operation necessary for the FFT appears to be novel.

The Quicksorting algorithm suitable for use on a cached, parallel, pipelined machine is likewise novel. The author learned of the Cray-1 Quicksorting program after he had developed the two algorithms described in chapter 8; the structure of the Cray-1 algorithm is similar to that of algorithm 1, while algorithm 2 seems to be entirely original.

Finally, the approach given in this thesis for an out-of-order pipelined execution structure which eliminates the problem of imprecise interrupts is apparently original, and should substantially enhance the average performance of processors which implement it.

## 11.3 Future Work.

The bulk of the obvious extensions of the present work consists of quantitative assessment of the impact of the individual and collective effects of the algorithms described in this dissertation on the performance of a digital computer employing them when processing various types of applications kernels and programs. This can be accomplished most readily through performance metering and subsequent analysis of the Mark IIA hardware in the course of actual program execution, and will be done by the author and his colleagues during the next year.

Longer range extension of the present work will involve studies of the throughput gains available from use of the out-of-order pipelined execution structure in the author's next computer implementation, and a much wider range of studies by the computing community of the performance levels which may be realized through exploitation of the techniques and algorithms described in this work, both in the S-1 Mark IIA and subsequent systems embodying them.

# 12. References

[Anderson67a] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, D. M. Powers, "Floating-Point Execution Unit", *IBM J. of Research and Development*, vol. 11, no. 1, pp. 34-53, January 1967.

[Anderson67] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, "Machine Philosophy and Instructions Handling", *IBM J. of Research and Development*, vol. 11, no. 1, pp. 8-24, January 1967.

[Budnik71] P. Budnik and D. Kuck, "The Organization and Use of Parallel Memories", *IEEE Trans. Computers*, vol. C-20, pp. 1566-1569, December 1971.

[Campbell62] S. G. Campbell, "Floating-point Operation", in *Planning a Computer System - Project Stretch*, W. Buchholz, ed., McGraw-Hill, 1962.

[Cotten65] L. W. Cotten, "Circuit Implementation of High-Speed Pipeline Systems", *AFIPS FJCC Proceedings* vol. 27, part 1, pp. 489-504, 1965.

[Cotten69] L. W. Cotten, "Maximum-Rate Pipeline Systems", *AFIPS Proc. 1969 SJCC*, pp. 581-586, 1969.

[Field69] J. A. Field, "Optimizing Floating-Point Arithmetic via Post Addition Shift Probabilities", *AFIPS Proc. 1969 SJCC,* pp. 597-603, 1969.

[Fike68] C. T. Fike, *Computer Evaluation of Mathematical Functions,* Prentice-Hall, 1968.

[Flynn70] M. J. Flynn, "On Division by Functional Iteration", *IEEE Trans. Computers,* vol. C-19, pp. 702-706, August 1970.

[Flynn72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Trans. Computers,* Vol. C-21, No. 9, pp. 948-960, September 1972.

[Hallin70] T. G. Hallin and M. J. Flynn, "Pipelining of Arithmetic Functions", *IEEE Trans. Computers,* vol. C-19, no. 8, pp. 880-886, August 1970.

[Holgate80] R. W. Holgate and R. N. Ibbett, "An Analysis of Instruction-Fetching Strategies in Pipelined Computers", *IEEE Trans. Computers,* vol. C-29, no. 4, pp. 325-329, April 1980.

[Kahan81] W. Kahan, private communication, March 1981.

[Knuth73] D. E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching,* pp. 114-123, Addison-Wesley, 1973.

[Knuth81] D. E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms,* 2nd ed., pp. 200-204, Addison-

Wesley,1981.

[Kogge77] P. M. Kogge, "The Microprogramming of Pipelined Systems", *Proc. Fourth Annual Sym. on Computer Architecture*, pp. 63-69, March 1977.

[Kogge81a] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, 1981.

[Kogge81b] Ibid., pp. 263-265.

[Kuck78] D. J. Kuck, *The Structure of Computers and Computations*, vol. one pp. 415-418, John Wiley & Sons, Inc., 1978.

[Kulsrud78] H. E. Kulsrud, R. Sedgewick, P. Smith, T. Szymanski, *Partition Sorting on Cray-1*, Communications Research Division Working Paper, SCAMP No. 7/78, *Institute for Defense Analysis*, Princeton, NJ, 1978.

[Lawrie75] D. H. Lawrie, "Access and Alignment in an Array Processor", *IEEE Trans. Computers*, vol. C-24, no 12, pp. 1145-1155, December, 1975.

[McWilliams79] T. M. McWilliams, *Verification of Timing Constraints on Large Digital Systems*, PhD. Thesis, Stanford University, May 1980.

[Morris79] D. Morris and R. N. Ibbett, *The MU5 Computer System*, Springer-Verlag, 1979.

[Polge74] R. J. Polge, B. K . Bhagavan, J. M. Carswell, "Fast Computational Algorithms for Bit Reversal", *IEEE Trans. Computers*, vol. C-23, no. 1, pp. 1-9, January 1974.

[Rabiner75] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, pp. 363-366, Prentice-Hall, 1975.

[Rabinowitz61] P. Rabinowitz, "Multiple Precision Division", *Commun. ACM*, vol. 4, p. 98, February 1961.

[Rau77] B. R. Rau and G. E. Rossman, "The Effect of Instruction Fetch Strategies upon the Performance of Pipelined Instruction Units", *Proc. Fourth Annual Sym. on Computer Architecture*, pp. 80-89, March 1977.

[S-1 Project 79] Staff, *S-1 Project FY1979 Annual Report*, University of California *Lawrence Livermore National Lab.*, UCID 18619, 1979.

[Sedgewick75] R. Sedgewick, *Quicksort*, PhD. Thesis, Stanford Computer Science Dept. Stan-CS-75-492, May 1975.

[Sedgewick78] R. Sedgewick, *Sorting on the Cray-1 : An Overview*, Communications Research Division Working Paper, SCAMP No. 16/78, Institute for Defense Analysis, Princeton, NJ, 1978.

[Shaham72] Z. Shaham and Z. R. Riesel, "A Note on Division Algorithms Based on Multiplication", *IEEE Trans. Computers*, vol. C-21, pp. 513-514, May 1972.

[Smith81] J. E. Smith, "A Study of Branch Prediction Strategies", *The Eighth Annual Symposium on Computer Architecture*, pp. 135-148, Computer Society Press, May 1981.

[Stephenson75] Stephenson, C. M., "Case Study of the Pipelined Arithmetic Unit for the TI Advanced Scientific Computer", *Proc. Third Annual Symp. on Computer Arith.*, pp. 168-173, 1975.

[Sweeney65] D. W. Sweeney, "An Analysis of Floating-point Arithmetic", *IBM Sys. J.*, vol. 4, pp. 31-42, 1965.

[Thornton70] J. E. Thornton, *Design of a Computer - the Control Data 6600*, pp. 77-88, Scott, Foresman and Co., 1970.

[Tomasulo67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM J. of Research and Development*, vol. 11, no. 1, pp. 25-33, January 1967.

[Winograd78] S. Winograd, "On Computing the Discrete Fourier Transform", *Math. Comput.*, vol. 32, pp. 175-199, 1978.

[Young73] D. M. Young and R. T. Gregory, *A Survey of Numerical Mathemetics, Vol. II*, pp. 1039-1062, Addision-Wesley, 1973.